Universidade de Santiago de Compostela

Departamento de Electrónica e Computación

TESIS DOCTORAL

OPENET4WF: MARCO DE CONOCIMIENTO PARA EL MODELADO DE FLUJOS DE TRABAJO FORMALIZADOS MEDIANTE REDES DE PETRI

Presentada por:
Juan Carlos Vidal Aguiar

Dirigida por:
Alberto J. Bugarín Diz
Manuel Lama Penín

Santiago de Compostela, Mayo de 2010

Dr. **Alberto J. Bugarín Diz**,
Profesor Titular de Universidad del Área de Ciencia de la Computación e Inteligencia Artificial de la Universidade de Santiago de Compostela

Dr. **Manuel Lama Penín**,
Profesor Contratado Doctor del Área de Ciencia de la Computación e Inteligencia Artificial de la Universidade de Santiago de Compostela

HACEN CONSTAR:

Que la memoria titulada **OPENET4WF: Marco de conocimiento para el modelado de flujos de trabajo formalizados mediante redes de Petri** ha sido realizada por D. **Juan Carlos Vidal Aguiar** bajo nuestra dirección en el Departamento de Electrónica e Computación de la Universidade de Santiago de Compostela, y constituye la Tesis que presenta para optar al grado de Doctor.

Santiago de Compostela, Mayo de 2010

Asdo: **Alberto J. Bugarín Diz**
Codirector de la tesis

Asdo: **Manuel Lama Penín**
Codirector de la tesis

Asdo: **Javier Díaz Bruguera**
Director del Departamento de Electrónica y Computación

Asdo: **Juan Carlos Vidal Aguiar**
Autor de la Tesis

Para mi familia, por su apoyo, paciencia y amor durante este largo camino.

Agradecimientos

Deseo expresar mi más sincero agradecimiento a todas las personas que, de alguna u otra forma, me han ayudado en la realización de este trabajo, y especialmente:

A Alberto J. Bugarín Diz y Manuel Lama, codirectores de esta tesis doctoral, por la confianza y ayuda que me han brindado a lo largo de estos años, sin las cuales la presente investigación no hubiera sido posible.

Al Departamento de Electrónica y Computación, y, principalmente, a los miembros del Grupo de Sistemas Inteligentes, por su apoyo y por los buenos ratos pasados. Recuerdo especialmente los buenos momentos que he pasado con mis compañeros de despacho Félix, Molina, Dinani, Daniel, Álvaro y Carlos.

A Reza Sadigh Balay por su colaboración y por el apoyo brindado a la hora de conseguir resultados en el difícil dominio de la fabricación de muebles.

A los amigos y compañeros que me han hecho especialmente agradable estos años de intenso trabajo. Y, por supuesto, a mi familia; sin ella nunca habría llegado hasta aquí.

Para finalizar, me gustaría agradecer el apoyo económico recibido para desarrollar esta investigación. Agradezco el soporte económico recibido de la Universidad de Santiago de Compostela a través de los proyectos “Sistema experto para la elaboración y seguimiento de ofertas en la industria del mueble” (PGIDT01INN35E) y “Sistema de planificación dinámica de la carga de trabajo de la fabricación en la industria del mueble” (PGIDIT04DPI096E); a la Xunta de Galicia a través del proyecto “Mellora da calidade educativa universitaria a través do deseño e execución de unidades de aprendizaxe soportados por tecnoloxías da Web Semántica” (PGIDIT06SIN20601PR); al Ministerio de Industria, Turismo y Comercio a través de los proyectos “SUMA elearning multimodal y adaptativo” (TSI-020301-2008-9) y “FLEXO: Desarrollo de aprendizaje adaptativo y accesible en sistemas de código abierto” (TSI-020301-2008-19) dentro del plan AVANZA; y del Ministerio de Ciencia e Innovación a través del proyecto “Descubrimiento automático de reglas borrosas no convencionales” (TIN2008-0040).

Mayo de 2010

El modo de dar una vez en el clavo es dar cien veces en la herradura.

Miguel de Unamuno

La victoria es del más perseverante.

Napoleón Bonaparte

Índice General

Índice General I

Índice de Figuras VII

Índice de Tablas XIII

Prefacio 1

1 Modelado de flujos de trabajo 7

1.1. La ambigua noción llamada “flujo de trabajo” 7
1.1.1. Distintos modelos para distintos propósitos 8
1.1.2. Caracterización de los flujos de trabajo 9
1.1.3. Clasificación de los flujos de trabajo 11
1.1.4. Lenguajes de modelado 13
1.2. Aproximación tradicional al modelado de flujos de trabajo 14
1.2.1. Sistemas de gestión de flujos de trabajo 14
1.2.2. Metamodelos de flujos de trabajo 15
1.2.3. Metamodelo tradicional de modelado de flujos de trabajo . . . 17
1.2.4. Problemas de la aproximación tradicional 19
1.3. Aproximación desde la perspectiva del conocimiento 21
1.3.1. ¿Qué es un sistema basado en conocimiento? 22
1.3.2. Metamodelos de métodos de resolución de problemas aplicados al modelado de flujos de trabajo 23
1.4. Conclusiones 32

2 Lenguajes de modelado de flujos de trabajo 37

2.1. Características de los lenguajes de flujos de trabajo 37
2.1.1. Expresividad del lenguaje 37
2.1.2. Legibilidad del modelo 52
2.1.3. Tipos de lenguajes 53
2.2. Formalismos de representación de los lenguajes de flujos de trabajo . . 54
2.2.1. Redes de Petri 54
2.2.2. Pi-calculus 59
2.2.3. Máquinas de estados abstractos 61

2.3. Lenguajes de representación 63
2.3.1. BPMN 63
2.3.2. Diagramas de actividad de UML 70
2.3.3. XPDL 73
2.4. Lenguajes de ejecución 76
2.4.1. BPML 76
2.4.2. BPEL4WS 79
2.4.3. YAWL 81
2.5. Lenguajes basados en tecnología semántica 83
2.5.1. OWL-S 84
2.5.2. WSMO 88
2.6. Conclusiones 91

3 Una ontología de redes de Petri para modelar procesos 95
3.1. Redes de Petri 95
3.1.1. Redes de Petri de bajo nivel 96
3.1.2. Redes de Petri de alto nivel 98
3.1.3. Redes de Petri de jerárquicas 101
3.2. Lenguajes de intercambio de redes de Petri de alto nivel 103
3.2.1. Desde el punto de vista del modelado 103
3.2.2. Desde el punto de vista tecnológico 104
3.3. Ontologías de redes de Petri 107
3.4. Ontología de HLPNs 109
3.4.1. Modelo estático 109
3.4.2. Modelo dinámico 121
3.4.3. Reglas 127
3.4.4. Compatibilidad con PNML 132
3.5. Ontología de HLPNs jerárquicas 136
3.5.1. Modelo de composición 136
3.5.2. Morfismo con red aplanada de alto nivel 141
3.5.3. Reglas 146
3.6. Conclusiones 149

4 Metamodelo del marco conceptual de flujos de trabajo 155
4.1. Arquitectura del metamodelo 155
4.1.1. Componentes de conocimiento 156
4.1.2. Uso de ontologías en el metamodelo 157
4.1.3. Ontologías que soportan la arquitectura del metamodelo 160
4.1.4. Adaptadores 161
4.1.5. Dimensiones de la arquitectura 164
4.2. Dimensión de procesos 164
4.2.1. Tareas 167
4.2.2. Métodos 169
4.2.3. Modelo de control 174
4.3. Dimensión de conocimiento del dominio 178
4.3.1. Modelo del dominio 179

4.4. Dimensión de recursos 181
4.4.1. Ontología de organización TOVE 182
4.4.2. Modelo de recursos 183
4.5. Puentes 185
4.5.1. Puente Tarea-Dominio 186
4.5.2. Puente Tarea-Método 187
4.5.3. Puente Método-Dominio 188
4.5.4. Puente Método-Control 189
4.5.5. Puente Método-Recursos 191
4.5.6. Puente Dominio-Control 192
4.5.7. Puente Dominio-Recursos 193
4.6. Refinadores 194
4.6.1. Refinador de tarea 195
4.6.2. Refinador de método 195
4.6.3. Refinador del modelo de control 197
4.6.4. Refinador del modelo del dominio 197
4.6.5. Refinador del modelo de recurso 198
4.6.6. Refinador de ontologías 199
4.7. Conclusiones 199

5 Modelo de control 203
5.1. Ontologías que soportan el modelo de control 203
5.2. Estructura del modelo de control 204
5.2.1. Propiedades que debería cumplir la red jerárquica 205
5.2.2. Características de la red jerárquica 206
5.2.3. Raíz de la red jerárquica 207
5.3. Patrón proceso 208
5.4. Patrones de flujos de trabajo 213
5.5. Mecanismos de composición de la red jerárquica 230
5.6. Anotación algebraica 233
5.6.1. Contexto de ejecución 234
5.6.2. Firma sintáctica 238
5.7. Conclusiones 241

6 Infraestructura para la ejecución de flujos de trabajo 243
6.1. Descripción de los flujos de trabajo 243
6.2. Composición del modelo de ejecución 243
6.2.1. Construir el modelo de ejecución 245
6.3. Gestor de mensajes 255
6.4. Participantes del flujo de trabajo 257
6.5. OPENET4WF: Motor de flujos de trabajo 258
6.5.1. OPENET: Motor de HLPNs 259
6.5.2. Capa de redes jerárquicas 265
6.5.3. Capa OPENET4WF 267
6.6. Conclusiones 268

7 Proceso de presupuestado en la industria del mueble 273
7.1. Automatización del presupuestado de muebles . . . 273
7.2. Proceso de elaboración de presupuestos . . . 275
7.2.1. Estructura . . . 275
7.2.2. Características . . . 276
7.3. Estimación de los tiempos de procesado . . . 279
7.4. Flujo de trabajo de presupuestado . . . 281
7.4.1. Presupuestado . . . 281
7.4.2. Establecer base de reglas de estimación de tiempos . . . 283
7.4.3. Describir detalladamente un pedido . . . 285
7.4.4. Diseño de producto . . . 286
7.4.5. Cálculo del coste de material . . . 287
7.4.6. Cálculo del coste de fabricación . . . 289
7.5. Arquitectura . . . 292
7.5.1. Integración del sistema de planificación . . . 296
7.5.2. Integración del sistema de aprendizaje de tiempos . . . 299
7.6. Evaluación del sistema . . . 300
7.7. Conclusiones . . . 303

8 Motor de ejecución de unidades de aprendizaje en IMS LD 307
8.1. Unidades de aprendizaje . . . 307
8.2. IMS Learning Design . . . 308
8.2.1. Niveles de implementación . . . 311
8.3. Motores de ejecución IMS LD . . . 312
8.4. Escenario de aprendizaje en astronomía . . . 314
8.5. Flujos de trabajo para la ejecución de unidades de aprendizaje . . . 316
8.5.1. Estructura de un método docente . . . 320
8.5.2. Estructura de una representación . . . 322
8.5.3. Estructura de un acto . . . 323
8.5.4. Estructura de una escena . . . 323
8.6. Arquitectura . . . 326
8.7. Conclusiones . . . 331

9 Motor de ejecución de servicios OWL-S 333
9.1. Estado del arte de la formalización de OWL-S mediante redes de Petri 333
9.2. Integración de OWL-S en el metamodelo de flujos de trabajo . . . 336
9.2.1. Integración del perfil del servicio . . . 337
9.2.2. Integración del modelo de servicios . . . 337
9.2.3. Integración del modelo de conexión del servicio . . . 354
9.3. Análisis de la coreografía de servicios OWL-S basada en redes de Petri 356
9.3.1. Validación de servicios OWL-S . . . 356
9.3.2. Verificación de servicios OWL-S . . . 357
9.4. Motor de orquestación de procesos OWL-S . . . 359
9.5. Conclusiones . . . 360

Conclusiones 365

A Aprendizaje de tiempos **371**
A.1. Estimación de tiempos de procesado en la industria del mueble 373
A.2. Polinomios de estimación del tiempo de procesado 374
A.3. Modelo de reglas TSK para la estimación de tiempos de procesado . . 376
A.3.1. Aprendizaje de bases de conocimiento TSK 377
A.3.2. Adaptación del modelo TSK 378
A.4. Método para el aprendizaje de tiempos 379
A.4.1. Descripción de la gramática libre de contexto 380
A.4.2. Algoritmo de programación genética 381
A.5. Resultados 386
A.5.1. Análisis de la precisión 388
A.5.2. Discusión acerca de la estructura del conocimiento 389

B Planificación de la producción **393**
B.1. Características del problema de planificación 394
B.2. Aproximaciones tradicionales 397
B.3. Adaptación del JSSP a la fabricación de muebles a medida 398
B.4. Método primitivo de planificación JSSP con máquinas dedicadas . . . 400
B.5. Método primitivo de planificación JSSP 403
B.5.1. Codificación del plan de trabajo 404
B.5.2. Generación del plan 406
B.5.3. Función de cruce 408
B.5.4. Función de mutación 409
B.5.5. Funciones objetivo 410
B.6. Resultados de la planificación 410

C Correspondencias entre OWL y FLORA-2 **417**

Bibliografía **423**

Índice de Figuras

1.1. Tipos de WFs en función del valor de negocio y repetitividad 12
1.2. Tipos de WFs en función de la estructuración y enfoque 12
1.3. Bucle característico de los metamodelos de comunicación 16
1.4. Dimensiones del metamodelo tradicional de definición de WFs 18
1.5. Ejemplo de clasificación de recursos por agrupaciones funcionales 20
1.6. Ejemplo de clasificación de recursos por roles 20
1.7. Diagrama de inferencia CommonKADS de un problema de control de calidad . . 25
1.8. Arquitectura de CommonKADS 26
1.9. Arquitectura de UPML 29
1.10. Interfaces de orquestación y coreografía de WSMO 30
1.11. Arquitectura conceptual de WSMF 31
1.12. Concepto de servicio en OWL-S 31
1.13. Metamodelo para la representación semántica de WFs en el que se introduce una nueva dimensión para el modelado del conocimiento en dominios complejos . . . 33

2.1. Patrones básicos de control. Cada rectángulo representa una actividad, las elipses representan construcciones de control y los arcos dirigidos representan las dependencias entre los distintos elementos. 40
2.2. Cuatro patrones avanzados de control 41
2.3. Ejemplo de un patrón para instanciación múltiple 42
2.4. Patrones de control basados en el estado 43
2.5. Ejemplo de tres patrones de cancelación 44
2.6. Patrones de terminación 45
2.7. Patrones que modelan estructuras iterativas 46
2.8. Patrones de control basado en eventos 46
2.9. Patrones de visibilidad 47
2.10. Patrón de interacción entre dos tareas 48
2.11. Ciclo de vida del trabajo 50
2.12. Estados del trabajo 52
2.13. Dos lenguajes de representación de WFs 53
2.14. Efecto de la visualización a la hora de interpretar dos modelos 53
2.15. Ejemplo de marcado, activación y disparo de una transición donde los círculos representan a las plazas, los rectángulos a las transiciones y las marcas se representan mediante puntos negros dentro de las plazas 55
2.16. Ejemplo de red de Petri coloreada 57
2.17. El patrón *mezcla simple* no puede resolverse con una transición normal 59

2.18. Solución al patrón *mezcla simple* utilizando arcos inhibidores 59
2.19. Representación de un bucle de tipo *ciclos arbitrarios* en π-calculus 61
2.20. Elementos básicos del lenguaje BPMN 64
2.21. Ejemplo simplificado de compra de una entrada de cine representada mediante un diagrama BPMN 65
2.22. Elementos básicos de los diagramas de actividad del lenguaje UML 71
2.23. Ejemplo simplificado de compra de una entrada de cine representada mediante un diagrama UML 72
2.24. Metamodelo de proceso de XPDL 74
2.25. Representación en XPDL de una separación 75
2.26. Metamodelo de proceso de BPML 77
2.27. Elementos básicos de YAWL 82
2.28. Taxonomía de OWL-S para la descripción de procesos 86
2.29. Ejemplo de integración entre el modelo de servicio y la capa de conocimiento de OWL-S 88
2.30. Orquestación y coreografía de WSMO 89

3.1. Principio de localidad y concurrencia de las acciones t_1 e t_2 97
3.2. Representación del ejemplo de fabricación mediante un grafo de PN 97
3.3. Disparo de las transiciones t_1 y t_2 98
3.4. PN coloreada del ejemplo de fabricación 99
3.5. Arcos anotados mediante multiconjuntos 99
3.6. Las transicións t_1 y t_2 se funden en la transición $t_{1,2}$ 100
3.7. Términos anotan arcos y transiciones 100
3.8. Ejemplo de sustitución (izquierda) y de fusión (derecha) 102
3.9. Ejemplo de red aplanada 102
3.10. Definición del concepto nodo en las especificaciones RELAX NG de PNML y FLORA-2 de la ontología 105
3.11. Definición de PN incorrecta a través de PNML 107
3.12. Red semántica del modelo estático de la ontología de HLPNs 110
3.13. PN que representa el estado inicial de una máquina de ensamblaje 113
3.14. Estado de la máquina de ensamblaje tras el disparo de la transición $t_{1,2}$ 113
3.15. Red semántica del modelo de situación de las HLPNs 123
3.16. Red semántica del modelo de ejecución de las HLPNs 124
3.17. Red semántica de la ontología de redes de Petri jerárquicas 137
3.18. Ejemplo de fusión de plazas 139
3.19. Refinamiento del proceso de preparación de las piezas para el procesado 142
3.20. Red semántica del morfismo que se define entre la red jerárquica compuesta de múltiples páginas y su la correspondiente red aplanada 144
3.21. PN de alto nivel aplanada resultante del mecanismo de sustitución representado en la Figura 3.19 146
3.22. Ejemplo de HLPN inicialmente creada para estimar el coste de fabricación de un mueble pero que se puede reutilizar en el dominio de la fabricación de piensos compuestos 150
3.23. Los agentes pueden compartir la información dinámica generada durante la ejecución de una HLPN 152

4.1. Marco de conocimiento para la descripción de WFs 156
4.2. El papel de las ontologías en la arquitectura del metamodelo 158
4.3. Ejemplo de una ontología de materiales 159

4.4. Pila de ontologías que describen la semántica del metamodelo 161
4.5. Relación entre los componentes de conocimiento y las dimensiones de los WFs . 165
4.6. Taxonomía de tarea genérica definida por Chandrasekaran [59] 166
4.7. Niveles de abstracción que intervienen en la definición de un WF 167
4.8. Taxonomía de tipos de problemas definida por la metodología KADS-1 [1, 228] . 168
4.9. Ejemplo de tarea de asignación de recursos 170
4.10. Ejemplo de método compuesto para la asignación de recursos 172
4.11. Ejemplo de método primitivo de selección 174
4.12. Ejemplo de diseño paramétrico usando HLPNs 175
4.13. Ejemplo de modelo de control 176
4.14. Representación gráfica del modelo de control de las figuras 4.13 y 4.15 177
4.15. Descripción parcial de algunas páginas del modelo de control de la Figura 4.13 . 178
4.16. Ejemplo de modelo del dominio de fabricación en la industria del mueble 180
4.17. Red semántica de la ontología de organización TOVE 183
4.18. Ejemplo de modelo de recursos 184
4.19. Ejemplo de puente entre una tarea y un modelo del dominio 186
4.20. Para establecer las correspondencias entre los conceptos *Plan* y *Agenda* de las taxonomías de la tarea y del dominio respectivamente, es necesario además definir la axiomática que permita unificar los distintos modos de establecer la duración de un programa 187
4.21. Ejemplo de puente entre una tarea y un método 188
4.22. Ejemplo de puente entre un método y un modelo de control 190
4.23. Parte de la descripción operacional del método descrito en la Figura 4.10 191
4.24. Ejemplo de puente entre un método y un modelo de recursos 192
4.25. Ejemplo de puente entre un modelo del dominio y un modelo de control 193
4.26. Ejemplo de puente entre un modelo del dominio y un modelo de recursos 194
4.27. Ejemplo de refinador que especializa el método compuesto representado en la Figura 4.10 196
4.28. Ejemplo de puesta en común de un modelo de control para así aprovechar mejor la capacidad de reutilización de los refinadores 198

5.1. Pila de ontologías que da soporte a la representación del modelo de control . . . 204
5.2. Patrón utilizado para la representación de un proceso 208
5.3. Ejemplo de aplicación del patrón *proceso* 213
5.4. Red semántica de la taxonomía de patrones de flujos de trabajo 214
5.5. Representación del patrón secuencia 215
5.6. Bloques de control del patrón secuencia 215
5.7. Representación del patrón separación 218
5.8. Representación del patrón sincronización 218
5.9. Separación y unión de dos hilos de ejecución 219
5.10. Bloques de control del patrón separación-unión 220
5.11. Representación del patrón elección 221
5.12. Bloques de control del patrón elección 221
5.13. Representación de una mezcla simple de dos ramas 222
5.14. Representación del patrón elección exclusiva-mezcla simpe para dos ramas . . . 224
5.15. Representación de la elección múltiple de dos ramas 225
5.16. Representación del patrón mezcla 226
5.17. Bloques de control del patrón mezcla 226
5.18. Representación de la *elección múltiple-mezcla múltiple* de dos ramas 227
5.19. Representación del patrón repetir-mientras 228

5.20. Bloques de control del patrón repetir-mientras 228
5.21. Representación del patrón repetir-hasta 230
5.22. Bloques de control del patrón repetir-hasta 230
5.23. Sustitución de la transición *scheduling* por un patrón *repetir-mientras* 232
5.24. Ejemplo de sustitución de construcciones de control 234
5.25. Ejemplo de red para el control del flujo de datos 235
5.26. Ejemplo de red para el control del flujo de datos teniendo en cuenta el principio de localidad 236
5.27. Ejemplo del problema de *lectura sucia* en una red que controla el flujo de datos . 236
5.28. Ejemplo de red que representa un flujo de control 236
5.29. Ejemplo de combinación de una red de datos y otra de control 237
5.30. Ejemplo de la segunda aproximación para el modelado de WFs 238
5.31. Ejemplo de un término que representa la llamada un operador 241

6.1. Arquitectura de la infraestructura tecnológica para la ejecución de WFs 244
6.2. Ejemplo de diagrama de descomposición de tareas 247
6.3. Integración del control dentro del diagrama de descomposición de tareas 248
6.4. HLPNs del modelo de control del método 1 249
6.5. Creación de la estructura jerárquica del modelo de ejecución 250
6.6. Precondiciones de un método primitivo en el álgebra del modelo de ejecución . . 251
6.7. Las transiciones que se resuelven mediante un método primitivo añadirán la precondición del recurso 252
6.8. Correspondencias entre los componentes de conocimiento y el álgebra de la red . 253
6.9. Los tres niveles de la anotación algebraica 255
6.10. Modelo de publicación/suscripción del repartidor (broker) de mensajes 257
6.11. Adaptador para la integración de servicios web 258
6.12. Adaptador para la integración de sistemas legados a través de CORBA 259
6.13. Arquitectura del motor OPENET 260
6.14. Ejemplo de ejecución de un multiconjunto de modos de transición 261
6.15. Un arco debe conectar a nodos de distinto tipo 262
6.16. Arquitectura del motor de flujos de trabajo 265
6.17. Regla para la transformación de una red jerárquica en una red plana 266
6.18. Regla para la unión de un conjunto de redes en una única red jerárquica 268
6.19. Regla para la creación de un patrón proceso (I) 269

7.1. Ciclo de vida para la tarea de presupuestado de muebles 276
7.2. Algunas de las uniones de madera más utilizadas 277
7.3. Descomposición en tareas del método que resuelve la tarea de presupuestado de muebles 282
7.4. Flujo de trabajo del método encargado de seleccionar la base de reglas de estimación de tiempos 284
7.5. Descomposición en tareas del método que resuelve la tarea encargada de crear la descripción del pedido a presupuestar 286
7.6. Descomposición en tareas del método que resuelve la tarea encargada del diseño del producto 287
7.7. Flujo de trabajo para el cálculo del coste de materiales 288
7.8. Flujo de trabajo para el cálculo del coste de producción 289
7.9. Flujo de trabajo para la generación del plan de trabajo 290
7.10. Flujo de trabajo para establecer el plan de trabajo 292
7.11. Arquitectura software de SEEPIM 293

7.12. Captura de la pantalla de modificación de una especificación de pedido 294
7.13. Captura de la pantalla del resumen de un presupuesto 297
7.14. Arquitectura del sistema de planificación 298
7.15. Captura de la pantalla de la carga de trabajo asociada a un pedido a través de un diagrama Gantt desde la interfaz web del sistema y exportada a Microsoft Project ™ 300
7.16. Esquema del sistema con la incorporación de los sistemas de planificación y aprendizaje al ciclo productivo de la empresa 301

8.1. El concepto de unidades de estudio en el EML de la OUNL [156] 309
8.2. Diagrama de actividades del escenario real de aprendizaje en astronomía 311
8.3. Estructura genérica que se encargará del control de la ejecución de los método docentes, representaciones, actos y actividades de las unidades de aprendizaje . . 317
8.4. Elementos que participan en la suspensión de un método docente 319
8.5. Flujo de trabajo que controla la ejecución de un conjunto de representaciones . . 321
8.6. Flujo de trabajo que controla la ejecución de una secuencia de actos 322
8.7. Flujo de trabajo que controla la ejecución de las entidades de ejecución 324
8.8. Flujo de trabajo que controla la ejecución de una actividad simple 325
8.9. Flujo de trabajo que controla la ejecución de una secuencia de entidades de ejecución 326
8.10. Flujo de trabajo que controla la selección de entidades de ejecución 327
8.11. Arquitectura orientada a servicios que externaliza las capacidades del motor de IMS LD nivel B 328
8.12. Arquitectura del motor de unidades de aprendizaje IMS LD nivel B 330

9.1. Integración de los servicios OWL-S dentro del marco de conocimiento 336
9.2. Patrón utilizado para la representación de un proceso en OWL-S 339
9.3. Ejemplo de aplicación del patrón *proceso* para un proceso de autenticación de usuarios 341
9.4. Red de Petri que define las salidas y efectos de un proceso en función del contexto de ejecución 341
9.5. Ejemplo de aplicación del patrón resultado para un proceso de autenticación de usuarios 344
9.6. Representación del patrón separación 345
9.7. Bloques de control del patrón separación 345
9.8. Representación del patrón elección 347
9.9. Bloques de control del patrón elección 347
9.10. Ejecución sin un orden predeterminado de dos procesos 349
9.11. Bloques de control del patrón sin-orden 349
9.12. Representación del patrón if-then-else 352
9.13. Bloques de control del patrón if-then-else 352
9.14. Pantallazo de la herramienta WoPeD utilizado para la verificación de las estructuras de control de los patrones *proceso* y *repetir-mientras* 357
9.15. Arquitectura del motor de coreografías OWL-S 360
9.16. Definición de un servicio de autenticación en OWL-S 361

A.1. Diseño conceptual a partir del cual se deducen las partes del mueble y los procesos de fabricación 374
A.2. Máquina lijadora calibradora 375
A.3. Gramática libre de contexto identificar los individuos válidos 381

A.4. Típica regla para la estimación de tiempos de procesado 381
A.5. Algoritmo evolutivo 382
A.6. Ejemplo de cromosoma para el aprendizaje de tiempos de fabricación 383
A.7. Ejemplo de cruce de dos cromosomas 385
A.8. Típica base de reglas para la máquina ACM 390

B.1. Diagrama Gantt de un plan de trabajo. El símbolo $O_{i,j}$ representa la operación j del trabajo J_i. 395
B.2. Diagram de Venn para los diferentes tipos de planes de trabajo 395
B.3. Diseño de una mesa de oficina 400
B.4. Representación en forma de red de Petri del método primitivo que resuelve el problema de planificación JSSP con máquinas dedicadas 401
B.5. Cálculo de la incertidumbre de una operación 402
B.6. Estructura del algoritmo NSGA-II [85] 404
B.7. Codificación de trabajo en paralelo de un ejemplo de planificación 4×4 (Primera parte del cromosoma) 405
B.8. Codificación de las reglas de asignación para el ejemplo de planificación 4×4 (Segunda parte del cromosoma) 405
B.9. Fechas límite, ordenación de las operaciones y tiempos de procesado 406
B.10. Método Giffer-Thompson modificado que se emplea para la generación del plan de trabajo 407
B.11. Ejemplo de algunos pasos de la ejecución del método modificado de Giffer-Thompson 408
B.12. Cruce de filas 409
B.13. Cruce de columnas 409
B.14. Frente Pareto final del NSGA-II y del SPEA-II para el problema *la24* 414

Índice de Tablas

1.1. Ventajas y desventajas de las distintas aproximaciones para el modelado de WFs 35

2.1. Posibles interpretaciones de plazas y transiciones en redes de Petri 55
2.2. Expresividad de los lenguajes de representación y ejecución de WFs (I) 66
2.3. Expresividad de los lenguajes de representación y ejecución de WFs (II) 67
2.4. Patrones de datos soportados por los lenguajes de representación y ejecución de WFs 68
2.5. Patrones de organización soportados por los lenguajes de representación y ejecución de WFs 69

3.1. Axiomas que restringen a la estructura del grafo. 114
3.2. Axiomas asociados a la firma sintáctica. 117
3.3. Axiomas que restringen a la anotación del grafo 119
3.4. Axiomas que restringen al álgebra de una HLPN 122
3.5. Axiomas que restringen al estado de la red 123
3.6. Axiomas que restringen a la ejecución de la red 128
3.7. Correspondencias entre atributos del PNML y la ontología de HLPNs 134
3.8. Axiomas que restringen las fusiones de nodos 140
3.9. Axiomas que restringen a las sustituciones de nodos 143
3.10. Axiomas que restringen el morfismo entre redes (I) 147
3.11. Axiomas que restringen el morfismo entre redes (II) 148
3.12. Capacidades de representación de las distintas ontologías de HLPNs 151

4.1. Ventajas y desventajas de UPML y OPENET4WF para el modelado de WFs . . 201

5.1. Axiomas que restringen la semántica del patrón *proceso* 210
5.2. Axiomas que restringen la estructura de una rama de control 216
5.3. Axiomas que restringen la semántica del patrón *secuencia* 216
5.4. Axiomas que restringen la semántica del patrón *separación* 217
5.5. Axiomas que restringen la semántica del patrón *sincronización* 219
5.6. Axiomas que restringen la semántica del patrón *separación-unión* 220
5.7. Axiomas que restringen la semántica del patrón *elección* 222
5.8. Axiomas que restringen la semántica del patrón *mezcla simple* 222
5.9. Axiomas que restringen la semántica de la *elección exclusiva-mezcla simple* . . . 223
5.10. Patrones de control avanzado soportados por los principales WMS comerciales . 224
5.11. Axiomas que restringen el patrón *elección múltiple* 225

5.12. Axiomas que restringen la semántica del patrón *mezcla múltiple* 226
5.13. Axiomas que restringen el patrón *elección múltiple-mezcla múltiple* 227
5.14. Axiomas que restringen la semántica del patrón *repetir-mientras* 229
5.15. Axiomas que restringen la semántica del patrón *repetir-hasta* 231
5.16. Axiomas que restringen la semántica del concepto `ExecuteSubstitution` 233
5.17. Axiomas que restringen la semántica del concepto `ControlSubstitution` 233

7.1. Variables a tener en cuenta en un proceso de fabricación 278
7.2. Variables que intervienen en la elección de un proveedor 278
7.3. Error medio en la estimación de tiempo por centro de coste 302

9.1. Axiomas que restringen la semántica del patrón *resultado* 342
9.2. Axiomas que restringen la *secuencia* 345
9.3. Axiomas que restringen la *separación* 346
9.4. Axiomas que restringen la *elección* 348
9.5. Axiomas que restringen la *separación-unión* 348
9.6. Axiomas que restringen las ramas concurrentes del patrón *sin-orden* 350
9.7. Axiomas que restringen el patrón *sin-orden* 351
9.8. Axiomas para la integración de la separación y sincronización con cada una de las ramas concurrentes 351
9.9. Axiomas que restringen la rama *si* del patrón *si-entonces-sino* 353
9.10. Axiomas que restringen al patrón *repetir-mientras* 354
9.11. Axiomas que restringen al patrón *repetir-hasta* 354
9.12. Verificación de las propiedades de OWL-S (I) 358
9.13. Verificación de las propiedades de OWL-S (II) 359

A.1. Error medio por centro de coste 372
A.2. Error medio por máquina 372
A.3. Características de los conjuntos de datos 387
A.4. Resultados de la validación cruzada para las distintas máquinas 389

B.1. Reglas de asignación de operaciones 406
B.2. Resumen de los resultados obtenidos para los métodos multi-objetivo 412
B.3. Comparación entre el NSGA-II y los demás métodos 413

C.1. Correspondencias entre OWL y FLORA-2 417

Prefacio

El uso de flujos de trabajo (WF, del ingés *Workflow*) es una constante en el quehacer cotidiano de cualquier empresa o administración y constituye un pilar fundamental en la organización del trabajo. Ya sea de manera implícita o explícita, los WFs están presentes en la mayoría de los procedimientos a los que diariamente estamos sometidos (por ejemplo, la reserva de un viaje, un parte por un incidente de tráfico, o la solicitud de un préstamo bancario). Es más, desde la perspectiva de un trabajador, varias de las tareas que realizamos a lo largo del día son parte de algún WF (por ejemplo, solicitar una ayuda para una conferencia, registrar la venta de un producto, o realizar el control de calidad de una pieza recién fabricada). Claro está que muchos de los WFs no están implementados y monitorizados a través de un sistema de gestión de WFs y en la actualidad muchos siguen siendo manuales. Sin embargo, no hay duda de que disponer de algún tipo de sistema con tecnología de WFs proporciona un conjunto de herramientas que facilitan notablemente la adaptación y mejora de nuestros procesos de negocio.

De los ejemplos anteriores se pueden intuir dos elementos diferenciados. Por una parte, la funcionalidad que aborda el WF y la coordinación de sus actividades, y por la otra, los agentes que participan en su ejecución y los criterios de asignación del trabajo. Otra característica muy importante, que suele pasar desapercibida, es la presencia de conocimiento, y con ello no nos referimos exclusivamente al conocimiento implícito en el WF (por ejemplo, la tarea de planificación no finaliza correctamente si no se asignan todos los procesos de fabricación a algún recursos) sino también a aquél que se utiliza para resolver una actividad en un determinado dominio de aplicación (por ejemplo, para el cálculo del tiempo de fabricación del seccionado se asume que siempre existen dos operadores de planta libres).

En la mayoría de los sistemas comerciales, este último tipo de conocimiento no se aplica, ya que consideran que los WFs tienen un alto nivel de granularidad y, por lo tanto, las actividades manejan su propio conocimiento. Sin embargo, cada vez son más frecuentes los WFs en los que distintas aplicaciones intercambian información. Para esos casos es necesario representar, al menos en parte, ese conocimiento para controlar su ejecución [314]. En lo que se refiere a este último punto, las soluciones existentes en la bibliografía son también divergentes tanto en lo que se refiere a la granularidad en la que se representan los datos [314] (por ejemplo, si se describen a nivel de documentos, objetos o simplemente parámetros) como a su visibilidad,

interacción o participación activa en el control del proceso [218]. En cualquier caso, ninguna propuesta aborda el manejo de esos datos desde la perspectiva de la gestión del conocimiento, y con ello se condiciona fuertemente su reutilización y la posibilidad de razonar acerca de sus características.

Partiendo de esta hipótesis, en esta memoria proponemos un metamodelo que permite integrar los elementos que forman parte de un WF dentro de una arquitectura de sistema basado en conocimiento (SBC), es decir, definimos un marco conceptual que facilita la representación de los procesos, de las estructuras organizativas y del conocimiento del dominio de forma explícita y a través un formalismo lógico para así mejorar la gestión y la reutilización del conocimiento del WF, así como el razonamiento semántico acerca de sus características. Para ello partimos de dos modelos, conocidos y aceptados, para, por una parte, representar WFs y, por la otra, modelar un SBC: las redes de Petri [203, 190] y los métodos de resolución de problemas [27] (PSM), respectivamente.

En esta memoria también presentamos el desarrollo del sistema OPENET4WF, un sistema de gestión de WFs construido sobre la base definida por el metamodelo. Esta característica convierte a OPENET4WF en un SBC ideal para WFs complejos y que hacen uso intensivo de conocimiento. En cualquier caso, la flexibilidad y escalabilidad de OPENET4WF no restringe el sistema a un tipo específico de WF, como atestigua su aplicación en el dominio de la industria del mueble, del aprendizaje electrónico, o de los servicios web semánticos.

Hipótesis

No existen en la actualidad sistemas de WFs que combinen las ventajas del modelado tradicional de WFs con las que proporciona la gestión del conocimiento. Estos sistemas tradicionales se centran exclusivamente en el diseño de los procesos de negocio y de sus actores sin considerar *(i)* el conocimiento implícito que se puede derivar de sus modelos, *(ii)* la forma de añadir nuevo conocimiento, *(iii)* reutilizar el conocimiento existente, y *(iv)* las características semánticas de sus componentes.

Objetivos

La principal finalidad de este trabajo es desarrollar e implementar un metamodelo para el modelado de WFs que permita capturar y reutilizar el conocimiento implícito en su definición y que además facilite el razonamiento acerca de sus características. Para ello planteamos una solución híbrida que combina las virtudes del modelado de procesos y las ventajas que proporcionan las arquitecturas de los sistemas basados en conocimiento a la hora de modelar y gestionar el conocimiento. Para lograr esta simbiosis, el desarrollo de este trabajo ha tenido que cubrir, además de las funcionalidades típicas de los sistemas de gestión de WFs, los siguientes objetivos:

1. Proporcionar una arquitectura conceptual para dar soporte al modelado de WFs a través de componentes de conocimiento independientes. El metamodelo que sigue esta arquitectura debe *(i)* facilitar la reutilización del conocimiento, y *(ii)* permitir razonar semánticamente acerca de las características de los WFs.

2. Integrar dentro del metamodelo y a través de una ontología la coordinación de las actividades de los WFs. El modelo de representación seleccionado ha de tener una alta capacidad expresiva y legibilidad dentro del metamodelo para así facilitar la definición de las estructuras de control complejas. Además, debe permitir razonar acerca del comportamiento del WF, lo cual indica que ha de tener la capacidad para razonar acerca de las características estructurales del WF y con ello poder responder a preguntas como ¿están ejecutándose estos dos procesos determinados en paralelo?, ¿de qué otro proceso depende un proceso dado?, ¿qué ejecución ha de producirse para poder alcanzar un determinado proceso?, etc.

3. Definir una ontología que facilite la creación y composición de WFs a partir de patrones o modelos de comportamiento conocidos y aceptados por los usuarios. De esta forma, puede verse el diseño de un WF como una tarea de diseño paramétrico donde se aboga por construir un modelo a partir de piezas independientes, como si se tratase de un rompecabezas.

4. Proporcionar un modelo de ejecución sin ambigüedades que se construya a partir de la combinación los distintos (e independientes) componentes de conocimiento del metamodelo, e implementar un motor de ejecución de WFs que interprete dicho modelo.

5. Validar nuestra propuesta mediante WFs con distintas características y en distintos dominios de aplicación.

Estructura de la memoria

Para dar validez a nuestro planteamiento y probar la consecución de sus objetivos, la organización de esta memoria es la siguiente:

- En el primer capítulo introducimos la problemática del modelado de los WFs, realizando una breve exposición de la forma tradicional en la que suele abordarse [264] e identificando las principales carencias de esta aproximación. El análisis demuestra que gran parte de las carencias se solucionan abordando el modelado de los WFs desde la perspectiva de la gestión del conocimiento. Finalizamos con una revisión de las principales propuestas planteadas en la bibliografía y el análisis de su aplicabilidad al modelado de WFs.

- En el segundo capítulo estudiamos los principales aspectos que han de modelarse en un WF y que definen la legibilidad y expresividad del metamodelo. Además, realizamos una revisión de los principales formalismos, lenguajes de representación y de ejecución de WFs.

- En el tercer capítulo introducimos el formalismo de las redes de Petri, que es el más empleado para el modelado de WFs debido a su gran capacidad expresiva y de legibilidad. Partiendo de este formalismo definimos una ontología de redes de Petri de alto nivel y otra de redes jerárquicas que usaremos dentro de nuestro metamodelo para representar la coordinación de las actividades de un WF. A través de estas ontologías la estructura y ejecución de los WFs se puede representar de forma explícita y sin pérdida de semántica.

- En el capítulo cuatro desarrollamos la arquitectura conceptual del metamodelo para el modelado de WFs. Esta arquitectura extiende el marco propuesto por UPML con dos nuevas dimensiones para manejar adecuadamente el control sobre las actividades y su coordinación con los recursos. Además, también definimos el conjunto de adaptadores que facilitan la composición de un modelo de WF y su reutilización.

- En el capítulo cinco profundizamos en el modelo de control del metamodelo propuesto, con el cual se coordinan las actividades de un WF. La base de este modelo es un conjunto de redes de Petri que capturan los patrones de comportamiento más relevantes de los WFs. Finalizamos el capítulo describiendo las principales características del álgebra que soporta las redes de Petri de este modelo.

- En el capítulo seis describimos la infraestructura que da soporte a la ejecución de los WFs. Más concretamente, explicamos cómo construir la red de Petri que representa el modelo de ejecución del WF, y finalizamos el capítulo presentando OPENET4WF, el motor que permite la ejecución de WFs.

- En los capítulos siete, ocho y nueve presentamos tres aplicaciones del metamodelo de WFs. La primera implementa la tarea de presupuestado en la industria del mueble. La segunda se orienta al dominio de la Educación y permite apreciar la versatilidad del metamodelo propuesto y cómo extender el motor OPENET4WF para dar soporte a la ejecución de unidades de aprendizaje. Finalmente, la tercera muestra cómo usar OPENET4WF para la ejecución de servicios web semánticos expresados en OWL-S.

- El último capítulo, se dedica a las conclusiones de la investigación realizada, resumiendo las aportaciones más importantes de este trabajo y mencionando algunos de los puntos abiertos más importantes que constituirán el trabajo futuro.

Además, en la memoria hemos incluido tres apéndices. En el primer apéndice desarrollamos uno de los métodos de resolución de problemas aplicado al problema del aprendizaje de tiempos en el dominio de la fabricación de muebles a medida. En el segundo apéndice planteamos el método de resolución de problemas para la planificación de recursos en una planta dedicada a la fabricación de muebles. En el último apéndice incluimos las correspondencias entre dos de los lenguajes de representación de ontología utilizados en esta memoria.

Publicaciones

Para finalizar, nos gustaría destacar que una parte importante de los resultados que se presentan en esta memoria han tenido su reflejo en diversas de publicaciones:

- Nuestra propuesta inicial para el modelado de WFs a través de SBCs [272] fue presentada en la *7th International Conference on Knowledge-Based Intelligent Information and Engineering Systems* (KES 2003) y una primera implementación de esta propuesta aplicada al problema de presupuestado en la industria del mueble [273] fue presentada en el *9th IEEE International Conference on Emerging Technologies and Factory Automation* (ETFA 2003) ese mismo año. Aunque existen significativas diferencias entre esta primera aproximación, basada en la metodología CommonKADS, y la presentada en esta memoria, ambos trabajos fueron el punto de partida de nuestra investigación.

- En el trabajo [267] presentado en la *10th International Conference on Knowledge-Based Intelligent Information and Engineering Systems* (KES 2006) se definen las bases arquitectónicas de nuestro metamodelo. Una versión extendida de este trabajo [271] fue presentada en el *AAAI 2008 Spring Symposium: AI Meets Business Rules and Process Management* (AAAI 2008) y constituye la base del Capítulo 4 de esta memoria.

- La ontología de redes de Petri de alto nivel que permite la representación de las redes descrita en el Capítulo 3 es una versión extendida del trabajo [266], publicado en la revista *Petri Nets Newletter*, y provee las bases para la representación de la coordinación de las actividades dentro del metamodelo propuesto.

- El núcleo de OPENET4WF es un motor que interpreta la ontología de redes de Petri, y ha sido publicado en la revista *Expert Systems with Applications* en el año 2010. Este motor soporta la interpretación del modelo de ejecución de WFs descrito en el Capítulo 6 y recogido en el trabajo [270] presentado en la *1st International Working Conference on Business Process and Services Computing* (BPSC 2007), el cual está a su vez basado en el trabajo publicado en las *III Jornadas Científico-Técnicas en Servicios Web y SOA* (JSWEB 2007).

- La aplicación del metamodelo de WFs al campo del presupuestado y fabricación de muebles a medida ha dado lugar a varias publicaciones. Una primera versión del sistema puede encontrarse en el capítulo de libro [268] perteneciente a la colección *Frontiers in Artificial Intelligence and Applications* de IOS Press. El modelo de WFs más detallado [275] fue presentado en la *23rd International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems* (IEA-AIE 2010).

- La resolución de la estimación de tiempos de fabricación [188] fue presentada en la *3rd International Symposium on Genetic and Evolving Fuzzy Systems* (EFS 2008). Una versión extendida de este artículo [189] fue publicada en la revista *Soft Computing*.

- La parte del modelo de presupuestado, que trata la planificación del presupuesto [284], fue presentada en la *2006 International Symposium on Genetic and Evolving Fuzzy Systems* (EFS 2006) y recibió el segundo premio al *best industry-orientated paper*.

- El problema de la asignación de recursos en la planta de producción [282] fue presentado en la *8th International Conference on Hybrid Intelligent Systems* (HIS 2008) consiguiendo el premio al *industry best paper application award*. Una versión extendida de este artículo [281] fue publicada en la revista *Applied Soft Computing*.

- Por su parte, en la *3rd European Conference on Technology Enhanced Learning* (ECTEL 2008) formalizamos la ejecución de unidades de aprendizaje en el campo de la educación a través de redes de Petri [279]. Una versión extendida de este artículo [235] donde las redes se anotan con la ontología de IMS LD ha sido publicada en la revista *Interactive Media in Education* [235].

- La implementación del motor OPENET LD ha sido publicada [285] en la *2008 European University Information Systems organisation* (EUNIS 2008) y una versión extendida [280] ha sido presentada en la *9th IEEE International Conferenceon Advanced Learning Technologies* (ICALT 2009). Una aplicación de OPENET LD a la ejecución de unidades de aprendizaje en entornos virtuales ha sido presentada [105] en las *V Jornadas Científico-Técnicas en Servicios Web y SOA* (JSWEB 2009) y una versión extendida [106] en la *10th IEEE International Conferenceon Advanced Learning Technologies* (ICALT 2010).

- Finalmente, la aplicación de OPENET4WF a la ejecución de servicios web semánticos también ha dado lugar a varias publicaciones. La aproximación inicial del modelo basado en redes de Petri [163] se presenta en las *II Jornadas Científico-Técnicas en Servicios Web y SOA* (JSWEB 2006). Una versión extendida donde se incluye la anotación de las redes [269] se presentó en el *1st International Workshop on Formal Aspects of Business Processes and Web Services* (FABPWS 2007).

1

Modelado de flujos de trabajo

Este capítulo introductorio aborda el modelado de flujos de trabajo desde una perspectiva estructural. Trata de ofrecer un análisis de las distintas características de los flujos de trabajo y de identificar cómo esas características se modelan y combinan dentro de los marcos conceptuales tradicionales. Este análisis está enfocado a los componentes de los flujos de trabajo y no a los aspectos metodológicos utilizados para identificar dichos componentes. Por ello, el núcleo de este capítulo consistirá en analizar los dos enfoques que mejor se adaptan al modelado de flujos de trabajo enriquecidos semánticamente: los sistemas de gestión de flujos de trabajo (WMSs, del inglés *Workflow Management Systems*) y los sistemas basados en conocimiento (SBCs).

1.1. La ambigua noción llamada "flujo de trabajo"

Desde la aparición del paradigma de servicios web, la palabra flujo de trabajo está de moda y es el término favorito para referirse a cualquier tipo de proceso. Sin embargo, a lo largo de su historia, este término ha tenido muchos significados. Algunas aproximaciones consideran que un flujo de trabajo es un proceso administrativo [97, 173], es decir, un proceso que se encarga de dar un servicio. Otras consideran que los términos "flujo de trabajo" y "proceso de negocio" son sinónimos [264]. En otros ámbitos, como el de servicios web, el término "flujo de trabajo" se suele usar para referirse exclusivamente a la dimensión de control de un proceso, es decir, a las dependencias entre los (sub)servicios que forman parte de un servicio compuesto [88]. Ante tanta disyuntiva algunos autores han optado por una definición más empírica y consideran que un flujo de trabajo es un proceso de negocio que está soportado por un sistema de gestión de flujos de trabajo [86]. Sin embargo, la definición más aceptada ha sido publicada por la Workflow Management Coalition[1] junto con un glosario con la terminología relacionada con los flujos de trabajo [294]. En ella se define un flujo de trabajo (WF, del inglés *Workflow*) como:

[1] http://www.wfmc.org

La automatización, completa o en parte, de un proceso de negocio durante la cual se define la secuencia de acciones, actividades o tareas utilizadas para la ejecución del proceso, incluyendo el conjunto de información intercambiada entre sus participantes.

De manera más informal, un WF indica aquello que va a suceder en el proceso de negocio y quiénes van a realizarlo. Su objetivo es, por lo tanto, garantizar que el trabajo se realice en el momento indicado y por el participante o recurso más adecuado. Para conseguir este objetivo, los WFs aproximan la construcción de procesos de negocio desde una perspectiva que busca pasar:

- De la programación a la composición.
- De la orientación a datos a la orientación a procesos.
- Del diseño al rediseño.

Este cambio de perspectiva también propicia un cambio en la terminología usada durante la definición de los procesos de negocio. Tradicionalmente se habla de entidades, objetos, funciones, tablas, etc. En cambio, con los WFs se habla de procesos, tareas, recursos, roles, unidades organizativas, reglas, etc.

1.1.1. Distintos modelos para distintos propósitos

La esencia de un WF es modelar un proceso de negocio para así determinar un orden en el trabajo a realizar, las responsabilidades del personal, la interacción entre los recursos, el intercambio de información, etc. Por ello, el objetivo del modelado de WFs es incorporar todos los aspectos relevantes del proceso al tiempo que se descartan otros irrelevantes. Sin embargo, el resultado del modelado dependerá en gran medida del propósito u objetivo del WF. Aunque la mayoría de los WFs se usan con el propósito de controlar la ejecución de un conjunto de tareas, existen muchas otras razones por las cuales se puede querer crear un WF [209]. Destacan las relacionadas con la comunicación (difusión y documentación), la puesta en práctica (simulación, análisis y ejecución) y la organización.

Es fácil imaginar que modelos de WFs que sirven para diferentes propósitos variarán tanto en los contenidos como en el detalle. Por ejemplo, un modelo de WF para el desarrollo de sistemas estará enfocado hacia la descripción de las tareas. Lo mismo sucede con los modelos usados para documentar cada uno de los pasos de la ejecución. En ellos cada paso define un conjunto de instrucciones que los recursos podrán consultar durante la ejecución del proceso de negocio. En cambio, un modelo de ejecución detallará tanto las tareas como el control sobre las mismas. Si el modelo se usa para la simulación deberá además incorporar la interacción con el cliente.

1.1.2. Caracterización de los flujos de trabajo

La caracterización de los WFs es también objeto de un gran debate. En función de la acepción o significado atribuido al término WF, se pueden encontrar en la bibliografía distintas opiniones acerca de cuáles son las características más relevantes, cómo llamarlas, qué comprende cada una de ellas o cómo cuantificar una determinada característica. En este apartado se describirán las cinco características más aceptadas por la comunidad investigadora de WFs [147]: *funcionalidad, comportamiento, información, operacionalidad* y *organización*. Sin embargo, existen clasificaciones alternativas donde la aproximación da más importancia a otros aspectos. Por ejemplo, en [226] se añade la perspectiva *temporal* para fijar las restricciones temporales ligadas a la ejecución de las actividades del WF. En este trabajo también se clasifican los WFs en función de sus características *transaccionales*. Este concepto también es recogido por otras clasificaciones [164, 30] para tratar las características transaccionales, que dependiendo de si los WFs son internos a una organización o entre distintas organizaciones pueden tener una solución diferente. Respecto a los WFs entre distintas organizaciones, en [30] también se añade como característica la *interacción* para reflejar las interdependencias entre los recursos y actividades de distintas organizaciones. En [164], los WFs se caracterizan por su control, datos, modelo de organización y transaccionalidad, pero también por su *granularidad* y *gestión de excepciones*. Además, el modelo de la organización está partido de forma que también se identifican como características principales a los *papeles* (roles) que los usuarios pueden tomar así como a los *compromisos* (*commitments*, en inglés) que pueden alcanzar. Finalmente, en [210] se considera que la clasificación propuesta en [147] es a demasiado bajo nivel por lo que se propone analizar las características de los WFs en función del *caso* que tratan, sus *mecanismos de enrutamiento*, su *asignación de recursos* y sus características de *ejecución*.

1.1.2.1. Funcionalidad

La perspectiva funcional responde a la cuestión *“qué hace el WF”* y suele incluir la descripción del objetivo del WF, sus datos de entrada y de salida, restricciones funcionales, así como la descomposición del WF en unidades de trabajo más pequeñas, que a su vez pueden ser actividades atómicas u otro (sub)WF.

1.1.2.2. Comportamiento

La perspectiva de comportamiento, también llamada perspectiva de proceso, responde a la cuestión *“cómo funciona el WF”* y se refiere al flujo de control entre las actividades del WF. Esta especificación es una parte esencial de un WF, ya que indica cómo coordinar las actividades mediante una serie de primitivas de control. Aunque estas primitivas varían de un sistema a otro, existe un conjunto básico que cualquier sistema debería soportar, en el que se incluye la secuencia, separación, sincronización, y ejecución condicional. En el capítulo 2 se describirán con más detalle las principales primitivas de control existentes en el campo de los WFs.

1.1.2.3. Tratamiento de la información

La perspectiva de la información contiene las estructuras de datos y el flujo de datos entre actividades. Su objetivo consiste en proporcionar el dato correcto en el tiempo indicado. Se pueden distinguir dos tipos de datos:

- *Datos de producción.* Son datos relevantes del proceso de negocio que el sistema utiliza para determinar las transiciones de una instancia de WF. Por ejemplo, datos usados dentro de las pre- y postcondiciones de las actividades o para realizar la asignación de trabajo.
- *Datos de control.* Son datos relevantes para gestionar el comportamiento del WF, pero que no tienen significado a nivel del proceso de negocio. Por ejemplo, los datos relativos a una instancia de WF.

Los WMSs especifican sus datos a través de parámetros, variables y el propio flujo de datos. Los *parámetros* se utilizan para pasar datos entre las actividades, y de acuerdo con [147], pueden ser sólo de lectura, sólo de escritura o de lectura/escritura. Además tienen asociado un ámbito de aplicabilidad, como por ejemplo una actividad o todo el WF. Al contrario que los parámetros, las *variables* son globales a todo el WF, es decir, si un usuario crea una variable entonces será accesible desde cualquier parte del WF. Finalmente, el *flujo de datos* relaciona los datos con las actividades indicando a qué parámetros y variables se acceden y desde qué actividad.

1.1.2.4. Operacionalidad

La operacionalidad responde a la pregunta *"cómo están implementadas las actividades"*. Sin embargo, la implementación de las operaciones del WF es transparente y está desacoplada del motor de WFs, ya que estará a cargo de aplicaciones (externas) que se integran dentro de los WMSs. Por ello, desde el punto de vista del diseñador la operacionalidad puede verse como una simple interfaz cuyos detalles de implementación carecen de importancia para el usuario. En cualquier caso, incluso prescindiendo de esos detalles, es necesario saber qué parámetros de entrada proporcionar, los parámetros a devolver, el modo de invocación, los códigos de error o incluso los detalles de la configuración de la aplicación externa a invocar.

1.1.2.5. Organización

La perspectiva de organización responde a la pregunta *"quién está a cargo de realizar el trabajo"* en un WF, es decir, permite definir las restricciones que dirigirán la selección del recurso más indicado para realizar cada una de las actividades del WF. Tanto en la bibliografía como en los WMSs existen enfoques diametralmente opuestos acerca de los aspectos que debería cubrir esta perspectiva. Mientras algunos sistemas y publicaciones ignoran completamente la organización, otros discuten en detalle los requerimientos de este componente. Por ejemplo, este aspecto es muy poco

relevante en los lenguajes de composición de servicios [213, 174] o en estándares de ejecución de WFs como BPEL [15] donde ni siquiera se trata. El problema radica en si se considera el aspecto organizativo como un elemento con entidad suficiente para ser modelado dentro del WF. Así, mientras algunos consideran que es un elemento imprescindible, aportando construcciones y desarrollando políticas de asignación de recursos [313, 255], otros proveen un soporte mínimo o incluso no llegan a definirlo [293]. Como se señala en [202], se suele describir la organización en base a tres aspectos:

- *Estructura organizativa.* Define los objetos de la institución, así como las relaciones entre esos objetos [147]. Por lo tanto, este aspecto es el encargado de especificar objetos como ’usuario’, ’rol’, ’grupo’, ’departamento’, división’, ’permiso’, etc. y relaciones del tipo ’supervisa’, ’pertenece a’, ’juega el papel’ o ’forma parte de’. Se trata pues de representar el esquema organizativo, así como la política de control y autorización de la institución.

- *Gestión de la asignación de trabajo.* Define la asignación del trabajo a los recursos y la forma de evaluar dicha asignación. De forma genérica se suele asignar un determinado trabajo a un conjunto dado de agentes, para lo cual existen muchos criterios de asignación. Los más aplicados son los basados en el rol que juega el agente dentro de la organización o en la división/departamento a la que pertenece, en las responsabilidades del agente, en sus habilidades, etc. El proceso de asignación en sí también puede variar. La mayoría de los WMSs asignan los recursos en tiempo de ejecución [224]. Sin embargo, algunos sistemas prefieren calcular la asignación en tiempo de diseño, con el consiguiente problema de tener que recompilar el WF para dar de alta nuevos agentes [224]. Otros utilizan una estrategia diferida [224], de forma que los recursos se calculan al iniciarse el WF.

- *Sincronización de la asignación de trabajo.* Define criterios de colaboración cuando el trabajo requiere o puede ser asignado a más de un recurso. En la mayoría de los WFs no es necesario indicar este aspecto. Sin embargo, en algunos contextos específicos puede resultar imprescindible. Por ejemplo, en [30] se enfatiza la necesidad de mecanismos de sincronización cuando la asignación de trabajo se realiza en WFs entre distintas organizaciones, y propone emplear dos estados de asignación adicionales denominados ’delegado’ y ’reenviado’ que se emplean en identificar qué trabajos han sido asignados a un socio externo y así no duplicar el trabajo.

1.1.3. Clasificación de los flujos de trabajo

El esquema de clasificación más aceptado en la comunidad de WFs [255] categoriza los WFs en función de su *valor de negocio* y de su *repetitividad*. Por una parte, el valor de negocio expresa la importancia del WF en la organización. Por la otra, la repetitividad indica la frecuencia en la que el WF se lleva a cabo. Debido a que la creación de un WF es económicamente costosa, la mayoría de las organizaciones se

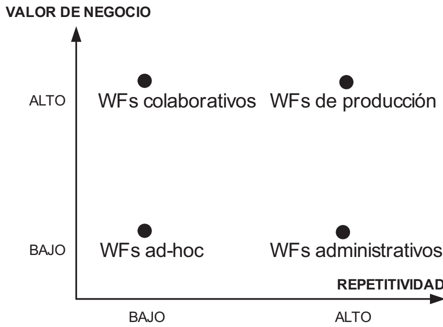


Figura 1.1: Tipos de WFs en función del valor de negocio y repetitividad

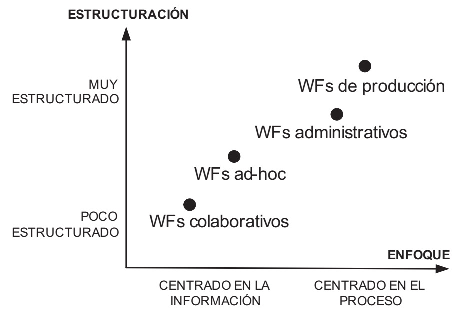


Figura 1.2: Tipos de WFs en función de la estructuración y enfoque

decantan habitualmente por automatizar los procesos de negocio que tengan un alto valor de negocio y una alta repetitividad. Las figuras 1.1 y 1.2 representan las cuatro clases de WFs identificadas:

- *WFs colaborativos y ad-hoc.* Son WFs de baja repetitividad que necesitan de la intervención humana para lograr la funcionalidad deseada. La principal diferencia entre ambas categorías está en su valor de negocio. Por una parte, los WFs colaborativos tienen un alto valor de negocio y prestan más atención a la comunicación entre los participantes y la compartición de información que a la definición del proceso. Este tipo de WF suele estar soportado por sistemas de trabajo en grupo (*groupware* en inglés), como Lotus Notes o Microsoft Exchange, más que por WMSs. Por otra parte, los WFs ad-hoc representan procesos donde el procedimiento no está definido de antemano, es decir, se crea y modifica durante la ejecución. Este tipo de WF tampoco suele estar soportado por WMSs [255].

- *WFs administrativos y de producción.* Son el tipo de WF soportado por la mayoría de los WMSs y se caracterizan por tener una alta repetitividad. Por un lado, los WFs administrativos son procesos bien delimitados y conocidos dentro de la organización. Suelen poseer un bajo valor de negocio y se desarrollan en dominios administrativos como por ejemplo el envío de un informe o la autorización de un proceso. Los WFs de producción suelen tener un alto valor de negocio, ya que se encargan de dar soporte al núcleo del negocio de la organización, como por ejemplo la gestión de reclamaciones en una compañía de seguros o la gestión de préstamos en un banco. Se caracterizan por ser muy estructurados y tener pocas variaciones a lo largo de su vida útil. En esta tesis, únicamente se considerarán estos dos tipos de WFs.

1.1.4. Lenguajes de modelado

Existen muchos lenguajes de modelado de WFs. Estos lenguajes difieren en sus construcciones de control, su notación, su uso, y muchos más aspectos. Algunos están orientados a la representación, otros a la ejecución; algunos son formales, otros ni siquiera tienen semántica operacional; etc. Dos razones justifican tanta variedad: el propósito del modelo y las características del WF.

1.1.4.1. Un lenguaje para cada propósito

Al igual que el propósito del WF influencia el contenido del modelo, el contenido del modelo hace lo propio con el lenguaje de modelado. Dependiendo de este contenido, algunos lenguajes son más indicados que otros:

- *Divulgación.* Los WFs orientados a divulgar información entre los participantes o a servir como soporte para la toma de decisiones suelen requerir de un lenguaje gráfico que facilite la interpretación de la información por parte de los participantes. Estos lenguajes centran el diseño en torno a las tareas y a sus participantes, pero también incorporan al modelo aspectos sociales y de comportamiento, de forma que sus modelos promuevan la conversación y la socialización de los WFs. En la práctica, este tipo de WFs suele utilizarse para introducir a los nuevos empleados a la estructura de los procesos de negocio, a los productos que distribuye o a las interdependencias inter-departamentales de la organización. Se caracterizan por ser sencillos y estar descritos a través de lenguajes informales [71].

- *Puesta en práctica.* Los diseñadores suelen utilizar la técnica propietaria proporcionada por el WMS en el cual están realizando el modelado. Ello es comprensible, ya que minimiza el esfuerzo de trasladar el modelo del WF a un modelo ejecutable. Sin embargo, el uso de técnicas propietarias en el modelado es una práctica desaconsejada debido a que *(i)* dificulta el análisis de las características y del comportamiento del modelo, ya que pocos de estos lenguajes disponen de modelos formales que faciliten el análisis del WF; *(ii)* dificulta la portabilidad de los modelos entre distintos WMSs (los WMSs deben compartir al menos un lenguaje común y ser la semántica del lenguaje equivalente); *(iii)* suelen ser difíciles de mantener.

- *Análisis.* Si el WF está diseñado con el propósito de analizar alguna faceta del modelo es imprescindible utilizar un lenguaje de modelado que disponga de técnicas de análisis. El análisis y la simulación son dos de los factores más importantes a la hora de seleccionar un lenguaje de representación [254]. Incluso si el modelado no requiere de estas capacidades, este tipo de lenguaje suele tener una base formal evitando así ambigüedades en el modelo. En la práctica no existen muchos lenguajes con este tipo de capacidades.

1.1.4.2. Muchas características y pocos lenguajes

La segunda razón que más influencia la selección de un determinado lenguaje para llevar a cabo el modelado concierne a las características del WF. Por ejemplo, un WF con un control complejo, que incluya tareas concurrentes, iteraciones o rutas condicionales, requiere de un lenguaje más expresivo que un WF que consista en una secuencia de tareas; si el WF se estructura en torno a las tareas, se modelará con lenguajes de modelado tradicionales; si lo hace en torno a la organización lo hará con técnicas que permitan la conversación y negociación entre los agentes; etc.

Independientemente de las características del WF, la gran mayoría de los WMSs permiten el modelado de WFs estructurados, y, por lo tanto, centrados en el control sobre las tareas. Además, la capacidad expresiva de sus lenguajes es suficiente para representar los WFs que se dan en el mundo real. Por ello, en la práctica las características tienen menos influencia en la técnica de modelado elegida.

1.2. Aproximación tradicional al modelado de flujos de trabajo

Aunque el concepto de WF no era nuevo, el impulso a esta tecnología no se produce hasta comienzos de los 90 cuando las organizaciones empezaron a prestar más atención a sus procesos de negocio y se dieron cuenta de las ventajas en cuanto a productividad, calidad del trabajo y reducción de costes de la automatización. En la actualidad, no se discute esta automatización, habida cuenta de que los procesos de negocio son cada vez más complejos, están sujetos a frecuentes cambios para adecuarse al mercado y el número de procesos que debe gestionar una organización es cada vez mayor.

1.2.1. Sistemas de gestión de flujos de trabajo

Un WMS es un paquete software que se puede usar para dar soporte a la definición, gestión y ejecución de WF [255]. En principio, la gestión de los WFs puede realizarse sin usar WMSs; de hecho, los WFs ya existían antes de la aparición de los WMSs y muchas organizaciones no necesitan de esta tecnología para implementar sus procesos de negocio. Sin embargo, a medida que los procedimientos se complican, la gestión del trabajo se hace insostenible sin un WMS.

La idea subyacente de los WMSs es la separación entre los procesos, los participantes y las aplicaciones. Por ello están enfocados a la logística del proceso de negocio y no al contenido de las tareas individuales. En la práctica, los WMSs se hacen cargo de entregar el trabajo a los recursos en tiempo y forma: cada vez que una pieza de trabajo ha sido completada durante la ejecución de un proceso de negocio, el WMS determina cómo continuar la ejecución y entrega la siguiente pieza de trabajo a realizar al recurso o recursos más adecuados. Los WMSs pueden asumir esta funcionalidad gracias al modelo del proceso de negocio en el que se definen todas las tareas así como sus dependencias. El WF también incorpora la información acerca del tipo de recurso

requerido para la ejecución de cada tarea, facilitando así la selección del recurso más indicado (normalmente una persona, un servicio o un sistema) por parte del WMS.

Al centrarse las empresas en el desarrollo de sus procesos de negocio, se han desarrollado a lo largo de los últimos años un conjunto de técnicas que son complementarias a las de gestión de WFs:

- Reingeniería de procesos de negocio (BPR, del inglés *Business Process Reengineering*). La BPR facilita el análisis y rediseño de procesos de negocio (y por lo tanto de WFs) dentro y entre organizaciones. BPR nace del trabajo [126] en el que se demuestra que en ocasiones cambios radicales en el diseño y organización de los procesos de negocio son necesarios para reducir costes e incrementar la calidad del servicio.

- Mejora continua de los procesos (CPI, del inglés *Continuous Process Improvement*). La CPI, al igual que la BPR, busca la mejora de los procesos de negocio pero mediante pequeñas mejoras en lugar de implementar un único paso con una gran mejora. Es una aproximación estratégica para promover una cultura de mejora continua en áreas como la fiabilidad, tiempos del proceso, costes, calidad o productividad. Se basa en la idea de que suele ser complicado encontrar una única causa de problemas en un proceso de negocio. Por ello, si no se está dispuesto a comenzar de nuevo el proceso de diseño, es mejor aplicar pequeñas mejoras dirigidas *(i)* eliminar las actividades sobrantes, *(ii)* reducir los tiempos de respuesta y *(iii)* simplificar el proceso o el diseño del producto.

- Gestión de logística (LM, del inglés *Logistics Management*). La logística se refiere al movimiento de mercancías, personas o energía desde un punto de origen hasta el cliente. La LM trata de optimizar cada parte del proceso de negocio para que el producto/funcionalidad llegue al cliente final de forma eficiente y en tiempo.

Estas tecnologías permiten cubrir otros aspectos relacionados con la mejora de los procesos que no son tratados por la tecnología de gestión de WFs (WM, del ingles *Workflow Management*). Sería, por lo tanto, incorrecto considerar a los WMSs como herramientas que se centran exclusivamente en la WM, ya que en la actualidad, los WMSs comerciales incorporan algunas de estas tecnologías con el fin de mejorar sus funcionalidades.

1.2.2. Metamodelos de flujos de trabajo

A pesar de los esfuerzos de los organismos internacionales, como la Workflow Management Coalition[2] (WfMC) o el Object Management Group[3], hasta el momento no se ha desarrollado ningún metamodelo genérico, pese a que hay un buen número de

[2] http://www.wfmc.org
[3] http://www.omg.org

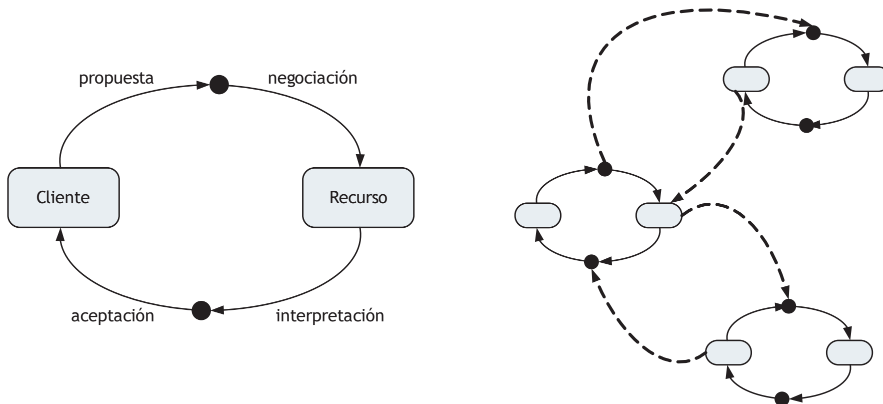


Figura 1.3: Bucle característico de los metamodelos de comunicación

ellos descritos en la bibliografía [52, 54, 58, 87, 94, 138, 146, 150, 159, 169, 264] y la mayoría de ellos han sido construidos en base a alguna de las características comentadas en el apartado 1.1.2. La clasificación de estos metamodelos en base a estas características puede encontrarse en [147]. Otra de las clasificaciones más aceptadas los categoriza en función de su enfoque:

- *Metamodelos de comunicación.* Nacen del modelo denominado "Conversation for Action Model" [297] y su objetivo es especificar la comunicación entre las entidades que participan en el WF. Los WFs se modelan como bucles donde el recurso realiza una acción bajo petición de un cliente (ver Figura 1.3) y cada bucle consta de cuatro fases consecutivas: *propuesta*, *negociación*, *interpretación* y *aceptación*. En la primera fase (propuesta) el cliente solicita la ejecución de una acción bajo unas determinadas condiciones; en la segunda fase (negociación) el cliente y el recurso acuerdan dichas condiciones; en la tercera fase (interpretación) el recurso ejecuta la acción bajo los términos acordados; y finalmente, en la última fase (aceptación) el cliente informa al recurso acerca de la satisfacción del resultado. La funcionalidad completa de un WF se consigue enlazando distintos bucles. Más ejemplos de este tipo de metamodelo de WFs pueden encontrarse en [176, 151].

- *Metamodelos de actividades.* Los metamodelos centrados en el flujo de control son hoy en día los más habituales y aceptados en la comunidad de WFs. Prueba de ello es la enorme cantidad de trabajos que en los últimos años han estado centrados en el modelado de procesos y de los que han surgido los actuales lenguajes de WFs [197, 10, 295, 247, 15, 259, 246, 168, 18, 129, 72]. En esta alternativa, la tarea/actividad es el elemento fundamental entorno al cual se crea el proceso. El control consiste en indicar el orden en el que se pueden ejecutar las tareas, sus condiciones de aplicabilidad y sus ejecutores. En esta tesis doctoral se ha optado por aproximar el modelado de WFs desde un metamodelo de actividades.

Otras clasificaciones de metamodelos centran su atención en el *paradigma/formalismo de representación* del WF. En este sentido, la mayor parte de estas clasificaciones están enfocadas en el flujo de control y prácticamente no consideran otras características como la organización o la información. Por ejemplo, en [141] se distinguen metamodelos basados en las *transiciones de estado*, en las *reglas* o en los *modelos imperativos*. En cambio, en [298] se clasifican en base a: *lenguajes de script*, *métodos orientados a redes*, *métodos basados en lógica*, *métodos algebraicos* y *reglas evento-condición-acción*. En [19, 11, 80, 164] pueden encontrarse otras clasificaciones similares a las anteriores. Finalmente cabe mencionar que, independientemente de sus características, es deber del metamodelo proporcionar un lenguaje que dé soporte a la definición del WF [164].

1.2.3. Metamodelo tradicional de modelado de flujos de trabajo

El diseño de un WF es una tarea complicada que requiere realizar un profundo análisis del proceso de negocio y de los recursos que participan en su ejecución. Tradicionalmente, esta tarea ha sido desarrollada por ingenieros del software y su realización ha estado enfocada hacia la definición de una estructura de procesos que cumpla con los objetivos del problema a resolver. Esta estructura actúa como elemento de integración de las distintas aplicaciones que se encargan de realizar alguna de las actividades del WF. La Figura 1.4 muestra este enfoque tradicional de construir un WF basado en un metamodelo de actividades. Puede verse claramente como un WF está compuesto por dos dimensiones [255]: una dimensión de *procesos*, que modela las relaciones entre las distintas actividades a realizar, y una dimensión de *recursos*, que organiza los agentes que intervienen en la ejecución de las actividades del WF. Un WF es, por lo tanto, el punto de encuentro entre estas dos dimensiones, es decir, el lugar donde los agentes se integran dentro de la estructura del proceso.

1.2.3.1. Dimensión de procesos

La dimensión de procesos determina la estructura de control del WF. El objetivo de esta dimensión es romper la estructura del proceso en partes más manejables, y el primer paso consiste en identificar los *(sub)procesos*, también denominados (sub)flujos, que encapsulan la complejidad del proceso en una relación jerárquica. Por ejemplo, un WF cuyo objetivo es la fabricación de un mueble podría dividirse en dos (sub)procesos: uno para la fabricación de cada una de las piezas y otro para el ensamblaje. La parte más pequeña de un proceso se denomina *tarea*, aunque también puede tomar el nombre paso, acción o actividad. Las tareas representan una unidad de trabajo cuyas características varían en cuanto al tiempo, espacio o requerimientos, pero todas suelen cumplir las propiedades transaccionales ACID; es decir, son atómicas, consistentes, independientes y duraderas. Por ejemplo, el encolado de dos piezas de madera podría considerarse como una tarea de un único paso donde la duración dependiese de las características del pegamento.

Dentro de un proceso de negocio típicamente existen *dependencias* entre tareas, donde una dependencia suele indicar un orden de ejecución a respetar. Por ejemplo,

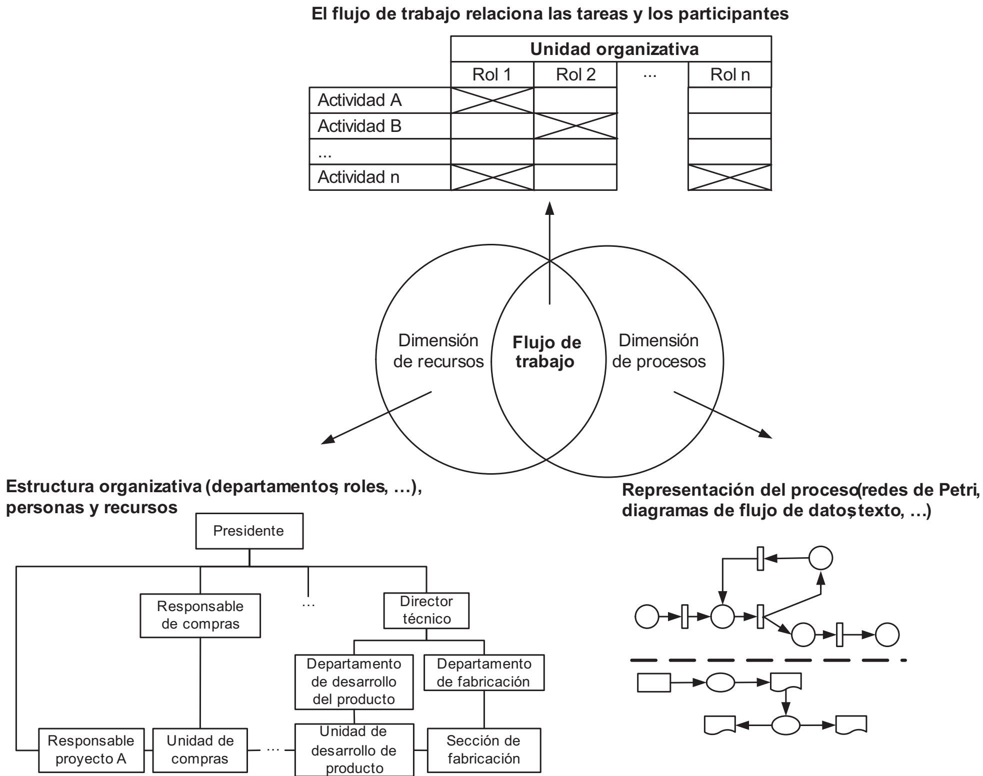


Figura 1.4: Dimensiones del metamodelo tradicional de definición de WFs

una dependencia puede usarse para garantizar que la tarea de encolado de dos piezas no se produzca antes de que las piezas hayan sido barnizadas. Aunque siempre expresen un orden, la semántica de una dependencia referencia el cumplimiento de una condición, un intercambio de información o un punto de control.

Otro elemento de la dimensión de procesos, además de las tareas y las dependencias, son los *componentes de control* que pueden considerarse como tareas de control, es decir, tareas cuyo objetivo no es la realización de una funcionalidad del proceso, sino la selección de las tareas en función de la instancia de ejecución. Existen muchos tipos de componentes de control, cada uno con un objetivo de enrutamiento diferente, algunos de los cuales son típicos de cualquier lenguaje de modelado de procesos o incluso de programación. Por ejemplo, entre los componentes de control más básicos se pueden encontrar la *secuencia*, *separación* (*split*, en inglés), *sincronización* (*join*, en ingles) o el conflicto. En el Capítulo 2 se estudiarán en detalle los componentes de control.

1.2.3.2. Dimensión de recursos

La dimensión de recursos trata los aspectos estructurales de la organización, es decir, define una estructura organizativa, clasifica los recursos dentro de esta estructura y asocia los permisos de cada uno de los recursos. La mayoría de los WMSs disponen de alguna herramienta que facilita el modelado de la organización, y aunque las características de estas herramientas difieren entre sí, típicamente tienen dos puntos en común: permiten definir y agrupar o clasificar recursos.

Los *recursos* identifican a un actor que está habilitado para ejecutar alguna de las tareas de un WF. Debido a la influencia de los sistemas de oficinas (*office systems*, en inglés), muchos investigadores consideran que los WFs modelan exclusivamente procesos administrativos dirigidos por personas, y por ello se refieren a los recursos como empleados o participantes [62]. Sin embargo, en la actualidad los WFs se utilizan para modelar cualquier tipo de proceso de negocio, desde la gestión de un almacén hasta un proceso de fabricación [257]. Por ello, el término recurso se refiere también a actores no humanos como máquinas, aplicaciones o servicios.

En el mundo real muchos negocios realizan una asignación de tareas a nivel de personas y donde cada tarea está asignada a un único recurso. El problema surge cuando el recurso no está disponible, bien sea por enfermedad, baja o avería. En estos casos es necesario reasignar manualmente la tarea a otro recurso, lo que implica modificar la definición del WF cada vez que se produzca este tipo de incidencias. Para evitar este problema y mejorar así la flexibilidad del sistema, los WMSs permiten organizar los recursos en distintas clases. En este contexto, una *clase* representa un grupo de recursos con similares características, como un departamento, una categoría profesional, una unidad, etc. La Figura 1.5 muestra un ejemplo de clasificación de recursos en el que se observa cómo un recurso puede ser miembro de más de una clase. Por ejemplo, *María* pertenece al *departamento de compras*, a la *sección internacional* y al *personal de soporte*.

Los recursos también suelen estar clasificados en función del rol que juegan dentro de la empresa, definiendo privilegios o permisos de acceso a la información y a la realización de las tareas. Esta clasificación suele combinarse con la estructura organizativa, creando así un segundo nivel transversal de clasificación, tal y como se muestra en la Figura 1.6.

Existen más clasificaciones aplicables al modelo de la organización en función de las habilidades, permisos, historial de asignaciones, carga de trabajo, etc. En el capítulo 2 se estudiarán algunos de los patrones de organización más utilizados y con ellos se podrá apreciar la extensión real de esta dimensión.

1.2.4. Problemas de la aproximación tradicional

El marco tradicional de modelado facilita la creación de un WF a partir de la estructura de sus procesos y de sus participantes. Sin embargo, únicamente con estas dos perspectivas la WM es limitada y adolece de:

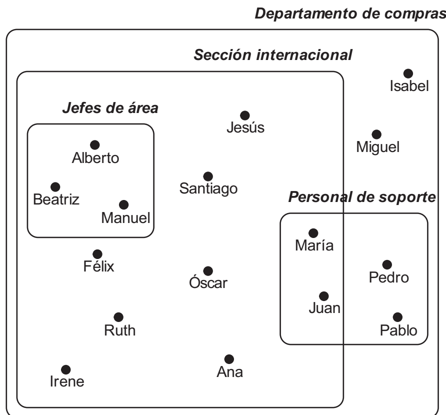


Figura 1.5: Ejemplo de clasificación de recursos por agrupaciones funcionales

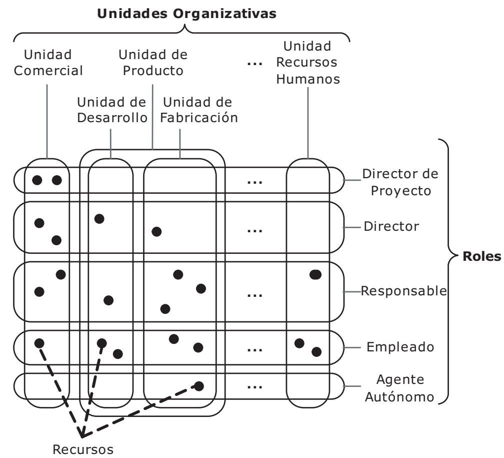


Figura 1.6: Ejemplo de clasificación de recursos por roles

- Una representación del proceso de negocio que trate cada una de *las componentes de un WF como una pieza de conocimiento* y, por lo tanto, permita preguntar al metamodelo acerca de su semántica [49, 130]. Estas preguntas pueden referirse tanto al modelo como a sus instancias, es decir, tanto a las características del WF en términos estructurales, de participantes, del dominio de la aplicación e incluso de las instancias actualmente en ejecución. En la mayoría de los WMSs la información se almacena en ficheros de texto o bases de datos con formatos que no siguen un paradigma lógico y que, por tanto, no están orientados al razonamiento. Además, los objetivos, parámetros y variables se especifican a gusto del diseñador y no tienen asociada ninguna semántica a la representación sintáctica. Por ello no es posible ni tan siquiera razonar acerca de las características funcionales de dos WFs, ya que los objetivos, parámetros de entrada y salida, o las condiciones no están representados semánticamente. En el mejor de los casos, se podría razonar acerca de los elementos estructurales o de control deducidos a partir del esquema del lenguaje de WFs. Sin embargo, este punto tampoco está garantizado al no estar basados todos los lenguajes en un formalismo de representación y, por lo tanto, permitir la ambigüedad.
- Un *modelo procesable* que facilite la automatización de la WM [130, 116]. La reducción (o eliminación) de la intervención humana es una característica deseable de cualquier WMS al igual que la inclusión de operaciones (semi)automáticas de composición, monitorización o recuperación de WFs. Si el WMS es entre distintas organizaciones entonces también cobra sentido la automatización de operaciones de descubrimiento, invocación o incluso negociación entre recursos.
- Un modelo que facilite la *reutilización del conocimiento del proceso* [17, 170]. La dimensión de procesos suele integrar la funcionalidad, comportamiento, información y operacionales del WF. Aunque de esta forma se facilita la definición

de un proceso a través de un único modelo, el modelo resultante resulta poco legible al integrar tanto los datos como las condiciones de enrutamiento o el código de las operaciones a aplicar. En estas circunstancias, se hace más costoso modificar o reutilizar el WF. En estas circunstancias, incluso resolver un caso relativamente sencillo como puede ser la reutilización de un WF por completo sin modificar su estructura puede tener sus complicaciones, ya que implica que el diseñador tenga que *(i)* analizar los tipos de datos empleados en el WF original *(ii)* sustituir esos datos por los correspondientes del nuevo dominio e incluso *(iii)* modificar el código del WF donde los datos estén integrados dentro de la estructura. En resumen, puede implicar reescribir el WF por completo manteniendo únicamente la estructura. Este problema se complica si la reutilización es sólo parcial: ¿cómo integrar los nuevos tipos de datos con los ya existentes? Por esta razón, la reutilización suele estar restringida a diseñadores expertos, hecho que no está alineado con las pretensiones de las herramientas de WFs, cuyo objetivo es requerir un bajo conocimiento informático a los usuarios. Para reducir los efectos de este inconveniente las interfaces gráficas de los WMSs suelen ocultar parte de la complejidad del procedimiento a los usuarios. Sin embargo, está claro que un mayor grado de parametrización en el metamodelo de WF, separando los distintos componentes que intervienen en su definición, mejoraría la reutilización sin necesidad de ocultar la complejidad real al usuario.

- Un modelo que facilite la *reutilización del conocimiento del dominio*. Aunque la visión tradicional del modelado de un WF puede ser válida para muchos dominios, no lo es tanto para dominios complejos donde el conocimiento experto tiene mucha influencia. En estos dominios suele existir una fase de adquisición de conocimiento paralela al diseño del WF y una fase en la que se integra el conocimiento adquirido dentro del WF. Esta solución tiene tres inconvenientes cuando se realiza desde una aproximación tradicional: *(i)* es costosa, *(ii)* propensa a errores y *(iii)* no facilita la reutilización del conocimiento adquirido. Los dos primeros problemas están relacionados con la fase de adquisición de conocimiento que, a pesar de la existencia multitud de herramientas, debe ser realizada por los desarrolladores de WFs debido a que dichas herramientas no son lo suficientemente intuitivas para que los expertos descarguen su conocimiento en ellas. Esto fuerza a los propios desarrolladores a convertirse en expertos del dominio a modelar, momento en el que se suele disparar la tasa de errores. Dado su elevado coste es, por lo tanto, imprescindible que dicho conocimiento se pueda aprovechar y reutilizar con facilidad en modelos de WF futuros. Sin embargo, al estar integrado dentro del modelo del WF, implicará una fase de extracción cuando llegue el momento de reutilizarlo.

1.3. Aproximación desde la perspectiva del conocimiento

Muchos WFs se caracterizan por tener una elevada complejidad y requerir la aplicación de conocimiento heurístico basado en la experiencia humana para alcanzar una solución. Los problemas con estas características se resuelven habitualmente median-

te *sistemas basados en conocimiento* (SBCs), que tratan de simplificar el problema mediante el uso de conocimiento adquirido directamente de humanos o automáticamente a través de técnicas abductivas, deductivas o inductivas. Este conocimiento se representa de forma declarativa (típicamente en reglas de producción), economizando así tanto en el proceso de desarrollo como de mantenimiento del sistema al tiempo que se mejora su comprensión y reutilización.

A pesar de todas estas ventajas, la mayoría de los sistemas de WFs comerciales no son SBCs ni disponen de mecanismos alternativos para incorporar conocimiento. La razón es sencilla: en general se considera demasiado costoso desarrollar WFs a través de los métodos de la ingeniería del conocimiento debido a que actualmente sigue existiendo una tendencia a identificar los SBCs como sistemas muy complejos y poco intuitivos, donde el conocimiento se codifica mediante reglas u otro formalismo de representación. Es más, desde la perspectiva de los diseñadores de WFs, esta visión de SBC es incompatible con la representación de los WFs, ya que la estructura del proceso queda oculta por el conjunto de reglas del modelo.

Sin embargo, los SBCs de "segunda generación" tratan de acabar con esta rigidez basando su desarrollo en técnicas de gestión del conocimiento (KM, del inglés *Knowledge Management*), y en particular en el uso de los *métodos de resolución de problemas* (PSMs, del inglés *Problem-Solving Methods*) [26, 27, 5], que en los últimos años han demostrado ser la principal alternativa para la descripción de procesos y que en la actualidad se están aplicando con éxito en el campo de los servicios web semánticos [83, 174]. Como se señala en [93, 155], la KM ha de jugar un papel fundamental en la evolución de los sistemas de WFs debido a que proporciona una visión global de lo que es un proceso de negocio. Si además se tiene en cuenta que los nuevos marcos conceptuales de KM se complementan con tecnologías semánticas, especialmente con el uso de ontologías, la combinación puede llegar a ser muy efectiva.

1.3.1. ¿Qué es un sistema basado en conocimiento?

Tradicionalmente, un SBC se define como un sistema computacional que utiliza conocimiento para resolver una tarea [241]. Sin embargo, a pesar de que muchos desarrollos son llamados SBCs, existe cierta controversia acerca de *qué considerar un SBC*. El principal problema radica en la palabra "conocimiento", ya que ésta se suele utilizar con distintas connotaciones. Por ejemplo, para calcular el coste real de un material, un sistema de compras en una empresa dedicada a la fabricación de muebles necesita como mínimo tener conocimiento acerca de las materias primas, de los materiales (pre)procesados, de los impuestos, del coste de almacenamiento, y del coste de amortización. Este sistema de compras podría verse como un SBC, y sin embargo, desde la perspectiva de la ingeniería del conocimiento sus características podrían no ser suficientes para tal consideración. Esto sugiere que la codificación explícita del conocimiento de una tarea o de un dominio no es suficiente para que un sistema sea considerado SBC, ya que también importa la forma de alcanzar la solución. Por ello, en [241] se indica que, en el contexto de los SBCs, la palabra "conocimiento" se refiere a la *experiencia codificada del agente*:

- *Experiencia.* Las soluciones a los problemas que intentan resolver la mayoría de los SBCs se caracterizan por su naturaleza experimental: es conocimiento aprendido en base a la experiencia o adquirido de fuentes externas.

- *Codificada.* Para que un sistema puede ser considerado "basado en conocimiento" debe necesariamente codificar de forma *explícita* el conocimiento manejado por el proceso. El énfasis en la representación del conocimiento ha sido históricamente uno de los puntos más importantes de los SBCs e incluso algunos autores [193] sostienen que es una condición *necesaria y suficiente* para que haya *inteligencia*.

- *Agente.* El término agente quiere enfatizar *(i)* que el conocimiento codificado sólo cobra valor cuando un agente es capaz de interpretarlo y *(ii)* que los SBCs usarán el conocimiento para realizar inferencias y obtener resultados.

1.3.2. Metamodelos de métodos de resolución de problemas aplicados al modelado de flujos de trabajo

Los SBCs originalmente usaban mecanismos de inferencia simples y genéricos para obtener las salidas a partir de casos particulares. Los motores de inferencia basados en la unificación, la resolución hacia delante o hacia atrás, o la subsunción cubrían la parte dinámica para derivar nueva información. Sin embargo, en el mundo real los expertos pueden explotar el conocimiento dinámico relacionado con la forma de resolver el problema. Partiendo de este hecho, en [70] se demostró el valor de dicho conocimiento a través de varios ejemplos donde los ingenieros del conocimiento implícitamente codificaron el conocimiento de control mediante la ordenación de reglas y la asunción de premisas acerca de dichas reglas de producción. El resultado fue una clara mejora del proceso de inferencia y la conclusión fue que hacer ese conocimiento explícito y darle el valor que le corresponde es una parte importante de un SBC y es la base en la que se apoyan los PSMs.

Los PSMs [5, 26, 27] refinan los mecanismos genéricos de inferencia anteriormente mencionados y permiten un una descripción más precisa del proceso de razonamiento. Concretamente, describen el conocimiento de control independientemente del dominio de la aplicación permitiendo así su reutilización en diferentes dominios y aplicaciones. Además, los PSMs se abstraen de un formalismo específico de representación en contraste con los motores de inferencia genéricos que dependen de una representación específica del conocimiento. Esta capacidad de representación del conocimiento dinámico y de reutilización es la que convierte a los PSMs en una alternativa viable para representar WFs y razonar acerca de los procesos de negocio.

En este apartado se describirán los principales metamodelos de PSMs aplicados al modelado de WFs. No se pretende hacer una descripción exhaustiva y en profundidad de las distintas metodologías y lenguajes existentes sino una introducción de las principales propuestas, remarcar los aspectos en los que se diferencian y analizar su idoneidad para modelar WFs. Esta revisión principiará con la metodología

CommonKADS, punto de partida de la mayoría de los metamodelos de conocimiento actuales, e introducirá cronológicamente los principales metamodelos de PSMs. Una especial atención merecen los dos últimos metamodelos descritos: WSMO [83] y OWL-S[4] [174]. Aunque ninguno de estos metamodelos han sido creado expresamente para representar WFs, la similitud existente entre algunos problemas del dominio de los servicios web y de los WFs hace que este modelado sea posible. Concretamente, la coreografía y la la orquestación de servicios web pueden verse como un problema de coordinación de procesos y, por lo tanto, como un problema de modelado de WFs. Aunque existen artículos teóricos que tratan el modelado de WFs con estos metamodelos, en la práctica únicamente existe un WMS implementado siguiendo la especificación OWL-S [24]. Este desarrollo se enmarca dentro del proyecto europeo NextGrid[5] cuyo objetivo es el desarrollo de la arquitectura de los futuros GRIDs. En cualquier caso, la falta de implementaciones no puede achacarse a la falta de interés sino a la juventud de ambas especificaciones. En el caso concreto de WSMO, a que algunas partes de la especificación todavía no están completas.

1.3.2.1. CommonKADS

La aproximación KADS para el modelado de conocimiento ha sido formulada a lo largo de más de una década (1983-1994) y ha dado lugar a dos especificaciones conocidas como KADS-I y KADS-II: KADS-I (1983-1989) propone una metodología para el desarrollo de SBCs; y KADS-II (1990-1994) se centra en el desarrollo de un metamodelo para el modelado de conocimiento. La revisión CommonKADS [228] dio lugar a una nueva formulación que integra el metamodelo de KADS-II dentro de la metodología KADS. Se puede considerar a CommonKADS como el fundamento teórico a partir del cual se desarrollaron los actuales PSMs.

Desde el punto de vista del modelado de procesos, CommonKADS distingue tres componentes principales para describir el conocimiento de una aplicación: *tarea*, *inferencia* y *dominio*. Las tareas describen *qué* se necesita para resolver el problema y *cómo* resolverlo. Su definición incluye las entradas, las salidas, los objetivos, la descomposición de la tarea y su control. Las inferencias describen las primitivas de razonamiento. Se consideran primitivas porque se describen exclusivamente desde el punto de vista funcional, ya que su estructura interna no es relevante desde la perspectiva del conocimiento. Finalmente, la última componente especifica la forma, estructura y contenidos del conocimiento relativo a un determinado dominio, que ha de ser relevante desde el punto de vista de la aplicación. Las relaciones entre los distintos componentes se establecen a través de los *roles de conocimiento*, es decir, a través de etiquetas abstractas que indican el papel que el conocimiento asociado a dicha etiqueta juega en el problema a resolver.

La Figura 1.7 muestra un ejemplo de diagrama de inferencia de CommonKADS

[4] OWL-S no es un metamodelo de PSMs sino una ontología para describir servicios web semánticos. Sin embargo, la forma en la que modela el control de los servicios está claramente fundamentada en la teoría de los PSMs y es el motivo por el cual se incluyó esta aproximación en este apartado.

[5] http://www.nextgrid.org/

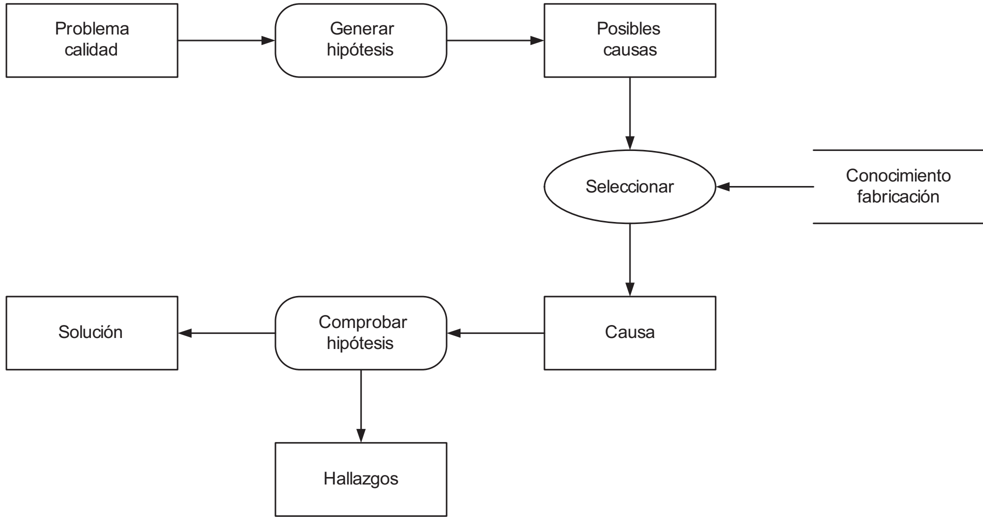


Figura 1.7: Diagrama de inferencia CommonKADS de un problema de control de calidad

en el que se distinguen estos tres tipos de componentes: las tareas están representadas mediante rectángulos redondeados, las inferencias mediante elipses y los roles de conocimiento mediante rectángulos simples si son roles dinámicos o rectángulos abiertos en el caso de ser roles estáticos. La estructura de inferencia de este ejemplo muestra la resolución de una tarea de control de calidad en un centro de fabricación. La fabricación de una pieza es un problema complejo y por ello cuando se detecta un fallo de calidad es necesario analizar las posibles causas: la tarea *generar hipótesis* se encarga de generar cada una de las posibles causas que pueden ocasionar tal defecto y la inferencia *seleccionar* de escoger la más probable a partir del conocimiento de fabricación; finalmente, se comprueba si la hipótesis se verifica. Este ejemplo permite apreciar cómo CommonKADS establece la relación entre las distintas tareas/inferencias a través de los roles de conocimiento. Estos diagramas de inferencia, sin embargo, no establecen el orden a la hora de ejecutar las tareas o inferencias que componen una tarea más genérica.

El principal aporte de CommonKADS al modelado de WFs sería *(i)* una mejor estructuración y reutilización de los datos, que se definirían a través de ontologías [122], *(ii)* la posibilidad de explicitar el conocimiento (normalmente a través de reglas) y *(iii)* una mayor capacidad de razonamiento acerca de las características funcionales de los procesos y de los modelos del dominio. Además, la metodología CommonKADS da soporte al modelado de la dimensión de recursos de los WFs. Como puede verse en la Figura 1.8, la metodología CommonKADS especifica un modelo de organización y un modelo de agentes a través de los cuales se puede capturar tanto la estructura como los recursos (humanos o software) de la organización. La integración entre los modelos de tareas, organización y agentes se realiza en los modelos de conocimiento

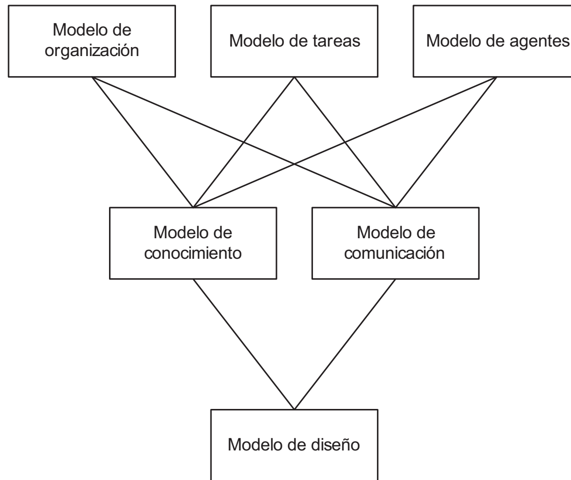


Figura 1.8: Arquitectura de CommonKADS

y comunicación.

Por el contrario, cabe destacar las carencias de CommonKADS a la hora de modelar el control sobre las tareas. CommonKADS sólo especifica la descomposición de tareas en (sub)tareas y relaciona dichas tareas a través de roles de conocimiento. Por lo tanto, conceptos básicos como la concurrencia, la sincronización o la secuencia no están contemplados en este marco de representación.

1.3.2.2. Componentes de la experiencia

Esta aproximación, denominada *Components of Expertise* en inglés, nació con el propósito de integrar un conjunto de propuestas esbozadas en los años ochenta en un marco de modelado comprensible [239]. La aproximación distingue tres componentes básicos de experiencia: *tareas*, *métodos* y *dominios*. La diferencia con respecto a CommonKADS es que las tareas sólo especifican aquello que debe resolverse y no la forma de resolverlo (esa parte del conocimiento está incluida en los métodos). Esta aproximación se caracteriza por introducir algunas ideas que luego fueron adoptadas por propuestas más modernas como puede ser Protégé II o, sobre todo, UPML. La principal es la separación entre tareas y métodos, o lo que es lo mismo entre funcionalidad y control. Esta separación permite resolver una misma tarea a través de métodos diferentes y, por lo tanto, reutilizar este elemento de control.

En cuanto al modelado de WFs tiene la ventaja, con respecto a CommonKADS, de separar los métodos de las tareas y así capturar mejor las perspectivas funcional y de control de los WFs. El principal inconveniente de esta aproximación es su cercanía al nivel de implementación, es decir, los métodos están descritos al nivel simbólico y no de conocimiento. Así, se reduce la capacidad de reutilización del modelo. En resumen,

aunque introduce mejoras respecto a CommonKADS, sigue teniendo inconvenientes a la hora de representar el control de los procesos. Además, no contempla la dimensión de recursos de los WFs.

1.3.2.3. Tareas genéricas

Como su nombre indica, esta aproximación [59] centra el modelado de conocimiento en torno a la noción de *tarea genérica* que integra tanto el concepto de tarea como el de método. Sin embargo, el enfoque de esta aproximación modela el conocimiento desde una perspectiva orientada a su uso: tanto la naturaleza como la representación del conocimiento del dominio están determinados por la tarea (o método), es decir, el conocimiento no se puede separar de su uso. Como consecuencia directa, esta aproximación sólo considera las tareas y los métodos. La relación entre métodos y tareas es también diferente: aunque las tareas siguen resolviéndose a través de métodos, estos últimos se descomponen en otras tareas cuando no son primitivos. Por lo tanto, la resolución de un método no primitivo no finalizará hasta que todas las (sub)tareas no se resuelvan a través de un método primitivo.

En cuanto al modelado de WFs, la descomposición de los métodos en (sub)tareas tiene la ventaja de permitir estructurar el control en base a las funcionalidades (o tareas) a resolver. De esta forma, se flexibiliza la definición de un WF, que ahora no tendría por qué especificar el control de sus (sub)procesos. Por lo demás, esta solución tiene los mismos inconvenientes que las aproximaciones tradicionales al modelado de los WFs, es decir, integra el conocimiento del dominio dentro de las tareas/métodos minimizando así su posible reutilización. Finalmente, esta solución tampoco aporta una descripción del flujo de control al nivel de abstracción requerido por un WMS ni contempla el modelado de la organización.

1.3.2.4. Protégé-II

La aproximación Protégé nació a partir del desarrollo de una herramienta de adquisición de conocimiento en el dominio de la medicina [206]. En su primera versión, el conocimiento se integraba desde la perspectiva de su uso y de forma similar a la propuesta de *tareas genéricas*. Sin embargo, los autores se dieron cuenta de las limitaciones de crear una herramienta exclusivamente para un determinado campo de aplicación y desarrollaron Protégé-II [113] con el fin de generalizar su uso a cualquier dominio y problema. Protégé-II reconoce tres tipos de componentes: *tareas*, *métodos* y *dominios*. En esta aproximación, los métodos se descomponen en (sub)tareas mientras que el término *mecanismo* se refiere a métodos directos. La principal diferencia con respecto a otras propuestas radica en que los métodos se describen en base a requerimientos ontológicos, relaciones de entrada y salida, flujo de control y flujo de datos. A pesar de fundamentarse en la perspectiva del *uso de conocimiento*, permite especificar métodos independientes del dominio y dominios independientes de los métodos para maximizar la reutilización. Para ello los métodos y dominios se integran a través de relaciones de correspondencias.

Esta aproximación tiene la ventaja de modelar explícita y claramente el control, aunque de modo insuficiente para dar soporte a todas las características de los WFs. Así, la descomposición de los métodos en (sub)tareas, la descripción de éstos a través de ontologías y la separación de los modelos del dominio de los métodos convierten esta aproximación en una de las más utilizadas para representar PSMs. Por último, mencionar que en propuesta la perspectiva de la organización no está explícitamente modelada, aunque podría considerarse su definición como una parte del modelo del dominio.

1.3.2.5. UPML

La aproximación UPML (del inglés *Unified Problem-solving Method description Language*) [196] nace como resultado del proyecto europeo IBROW [28] cuyo objetivo era el desarrollo de un servicio de consulta inteligente que permitiese recuperar componentes de conocimiento de librerías digitales distribuidas. UPML proporciona la arquitectura conceptual de IBROW y desde entonces se ha utilizado para el desarrollo de sistemas de razonamiento que hacen uso intensivo de conocimiento.

UPML modela PSMs a través de componentes de conocimiento y de adaptadores. Los primeros se refieren a cada una de las piezas del nivel de conocimiento necesarias para describir un PSM. Específicamente, distingue cuatro tipos de componentes de conocimiento que son independientes entre sí: *tareas*, *métodos*, *modelos de dominio* y *ontologías*. Por un lado, las *tareas* detallan el problema que resuelve el PSM a través de la descripción de sus propiedades funcionales, es decir, sus objetivos, entradas, salidas, pre- y postcondiciones. Los *métodos* representan la solución a las tareas. Al describirse en el nivel de conocimiento no implementan una solución a nivel simbólico: en el caso de métodos primitivos simplemente se indican sus propiedades funcionales; mientras que si son métodos compuestos, detallan el control sobre las (sub)tareas que lo componen. Los *modelos del dominio* permiten describir el conocimiento del dominio de la aplicación a través del conjunto de hechos, reglas, propiedades y asunciones que lo caracterizan. Finalmente, las *ontologías* son el componente central del metamodelo UPML y aportan el vocabulario y la axiomática necesaria para describir la semántica de los tres componentes anteriormente citados. De esta forma, cualquier tarea, método o modelo del dominio se describirá a través de una o varias ontologías, facilitando así su descripción y reutilización.

Como se muestra en la Figura 1.9 las tareas, métodos, modelos del dominio y ontologías se relacionan a través de un conjunto de adaptadores. Los *adaptadores* son relaciones binarias entre dos componentes de conocimiento: si la adaptación se establece entre componentes del mismo tipo, el adaptador se denomina *refinador*; en cambio, si se especifica entre componentes de distinto tipo se conoce como *puente*. Como su nombre indica, los *refinadores* permiten definir un nuevo componente añadiendo restricciones a la definición de otro componente preexistente. En cambio, los puentes se utilizan para adaptar la terminología empleada por dos componentes de conocimiento de forma que hablen el mismo lenguaje. Esta adaptación es sencilla si se puede asociar directamente el vocabulario de las ontologías que describen cada

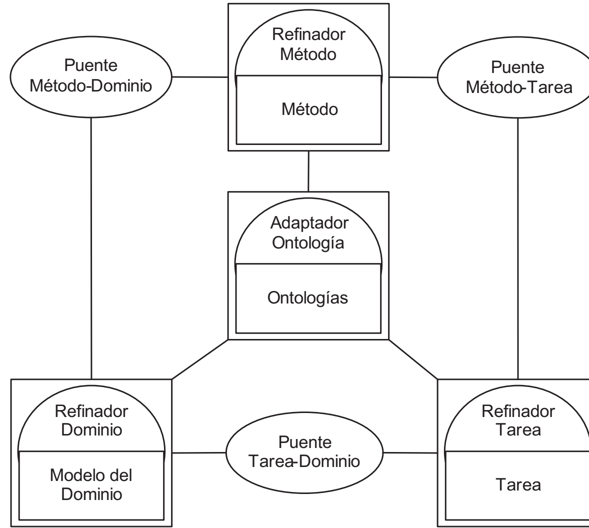


Figura 1.9: Arquitectura de UPML

uno de los componentes de conocimiento, o más compleja si requiere la definición de axiomas para ello.

La descripción de un WF a través de UPML permitiría subir un nivel más el ratio de reutilización, ya que en este marco todo es reutilizable. Respecto al modelado de la perspectiva de control, la propuesta es similar a la de Protégé-II, es decir, se distinguen dos tipos de métodos en función de si se descomponen en (sub)tareas o no y se especifica un control en un lenguaje que se define en el nivel del conocimiento y que, por lo tanto, no es procesable ni interpretable. Por lo tanto, la expresividad del control sigue siendo muy reducida comparada con los modelos de WFs. La perspectiva de la organización y de los recursos tampoco tiene un soporte directo, aunque al igual que en Protégé-II se podrían definir dentro del modelo del dominio a través de reglas de asignación y de ontologías.

1.3.2.6. WSMO

La aproximación WSMO (Web Services Modeling Ontology) [83] nace como uno los esfuerzos de la web semántica para representar el conocimiento dentro de la web. Para tal tarea, WSMO define una arquitectura denominada WSMF [103] (del inglés *Web Services Management Framework*) que proporciona un modelo conceptual para desarrollar y describir servicios web. Se basa en dos principios complementarios: *fuerte desacoplamiento* de los múltiples componentes que describen una aplicación de comercio electrónico y fuerte servicio de *mediación*, que permite a cualquier aplicación comunicarse con otros servicios y aplicaciones de una manera escalable. La arquitectura conceptual de WSMF está representada en la Figura 1.11 y se estructura entorno

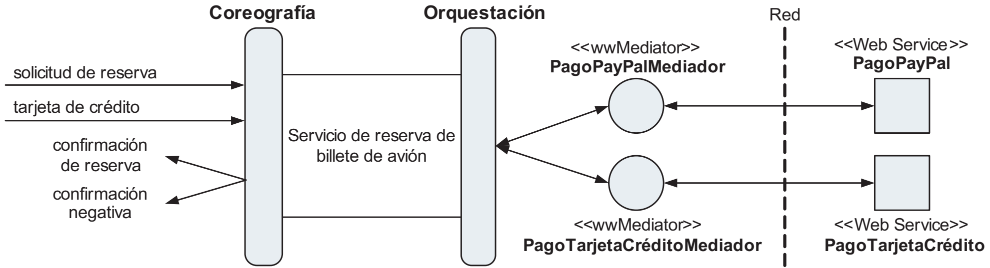


Figura 1.10: Interfaces de orquestación y coreografía de WSMO

a cuatro componentes:

- *Ontologías.* Permiten especificar formalmente la terminología usada por los demás componentes del metamodelo.
- *Objetivos.* Especifican los objetivos que el cliente debe poseer cuando consulta un servicio web. Se trata de especificar la capacidad del servicio a través de propiedades no funcionales, además de las pre- y postcondiciones, las asunciones y los efectos que un servicio web debe tener para cumplir con el comportamiento deseado.
- *Servicios web.* Describe lo que es un servicio web, sus características funcionales, no funcionales coreografía y orquestación. La Figura 1.10 muestra los dos tipos de interfaces que permiten componer un servicio web. Por una parte la *coreografía* describe la interacción del cliente a la hora de consumir el servicio: el proceso detalla los mensajes enviados y recibidos entre el cliente y el servicio, y también la tecnología de conexión y protocolos necesarios para que esta interacción tenga lugar. Por otra parte, la *orquestación* describe la forma de lograr la funcionalidad del servicio mediante la agregación de otros servicios web, y para ello, descompone la funcionalidad en (sub)funcionalidades.
- *Mediadores.* Permiten un fuerte desacoplamiento de los componentes de WSMO que describen los servicios web y se utilizan para la intermediación entre dichos componentes. Los mediadores facilitan la reutilización de cualquiera de los componentes WSMO en diferentes dominios y aplicaciones.

Al contrario que los otros metamodelos de PSMs, WSMO describe el control a través de un formalismo de representación de procesos: máquinas abstractas con estado [124]. Sin embargo, como está explicado en el Capítulo 2, este formalismo no es el más adecuado para representar WFs debido a que no introduce construcciones de control con las que se abstrae la definición de la dimensión de procesos, permitiendo un mayor entendimiento del proceso y una mayor legibilidad. En otras palabras, en WSMO la descripción del control se realiza a bajo nivel de abstracción.

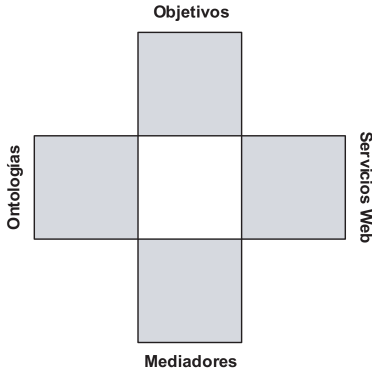


Figura 1.11: Arquitectura conceptual de WSMF

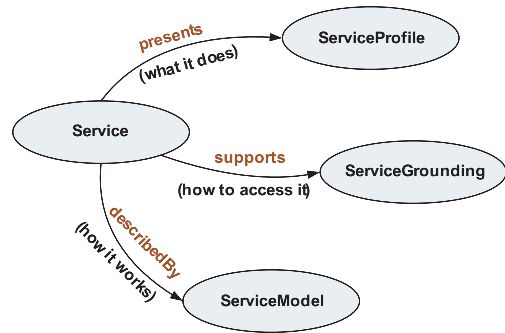


Figura 1.12: Concepto de servicio en OWL-S

El nivel de reutilización de componentes de WSMF es muy alto, sin embargo, al ser un metamodelo pensado para servicios web restringe su aplicabilidad a los WFs. Un ejemplo de ello es que WSMF no explicita la dimensión de recursos. En cualquier caso, WSMO permitiría definir implícitamente esta dimensión a través de las ontologías que capturan el conocimiento del dominio pero a costa de una menor capacidad de reutilización.

1.3.2.7. OWL-S

Al igual que WSMO, OWL-S [174] surge de la necesidad de representar semánticamente los servicios web tradicionales para así facilitar la realización de operaciones de descubrimiento, selección, composición, negociación, invocación, monitorización y recuperación de forma (semi)automática. Un servicio web semántico está formado por un servicio web y una anotación semántica, que consiste en asociar conceptos y relaciones de una ontología especificada en el lenguaje OWL [84] con los parámetros y operaciones de dicho servicio. Al contrario que WSMO, OWL-S no posee una arquitectura conceptual que defina los componentes necesarios para realizar dichas operaciones sobre los servicios web semánticos.

Como se puede ver en la Figura 1.12, OWL-S especifica a través de tres modelos la estructura de un servicio web:

- El *perfil del servicios* (*Service Profile*, en inglés) responde a la cuestión *“qué hace el servicio”* describiendo sus entradas/salidas, sus características funcionales, sus condiciones de aplicabilidad (pre/post-condiciones), y algunas características no funcionales, como es la calidad de servicio o la localización geográfica.
- El *modelo de conexión* (*Service Grounding*, en inglés) responde a la cuestión *“cómo puede un agente acceder al servicio”* describiendo los detalles del pro-

tocolo de comunicaciones, formato de los mensajes y puertos de comunicación que son usados para invocar el servicio.

- El *modelo de servicios* (*Service Model*, en inglés) responde a la pregunta *"cómo funciona el servicio"* detallando al cliente las condiciones bajo las cuales un determinado resultado puede producirse y la coreografía de mensajes que debe tener lugar con el servicio para obtener dicho resultado.

Desde el punto de vista del modelado, la estructura del proceso se establece en el modelo de servicios. OWL-S estructura el servicio en torno al concepto de *proceso*, que puede ser *atómico* o *compuesto* en el caso de descomponerse en (sub)procesos. Además, al contrario de WSMO, OWL-S proporciona estructuras de control predefinidas para así facilitar la composición de los (sub)procesos. Por ejemplo, define *secuencias* de procesos, procesos en *paralelo*, *sincronización* de procesos, y estructuras iterativas o condicionales.

El uso de OWL-S para representar WFs es, cuanto menos, mucho más sencillo que con WSMO. En este sentido, aunque OWL-S ha sido creado para representar servicios, éstos se describen como procesos. La estructuración de estos procesos es también más aplicable, ya que OWL-S proporciona un conjunto de construcciones de control, la mayoría de las cuales son típicas de los WFs.

Por otra parte, la especificación OWL-S tiene la desventaja de que no ha sido creada a partir de un modelo formal y, por lo tanto, tiene una semántica operacional ambigua. Aunque se han propuestos algunos modelos que han intentado dotar de un soporte formal a OWL-S [191, 89, 34, 308, 64, 178, 14, 183, 81] ninguno de ellos tiene en cuenta todos los elementos de la especificación. En esta tesis doctoral se ha dado respuesta a esta cuestión desarrollando un modelo formal basado en redes de Petri que atiende a todos los aspectos de OWL-S [277].

La reutilización tampoco es el punto fuerte de OWL-S, ya que no proporciona el marco adecuado para ello. En este sentido, no indica cómo reutilizar el modelo de conexión o cómo reutilizar un (sub)servicio. Este hecho está en gran medida relacionado con la falta de un marco arquitectónico que permita relacionar todos los componentes de OWL-S.

Desde la perspectiva de los recursos, OWL-S tampoco proporciona el marco adecuado para su modelado: aunque permite modelar recursos a través de una ontología de actores, esta ontología no está pensada para clasificar los recursos como requieren los WFs.

1.4. Conclusiones

En este capítulo se ha profundizado en el concepto de WF y se han introducido algunas de las muchas variantes a partir de las cuales se puede enfocar su modelado. En particular, se ha ahondado en el marco de modelado tradicional, utilizado por la mayoría de los WMSs, donde la descripción del WF se centra en su modelo de control.

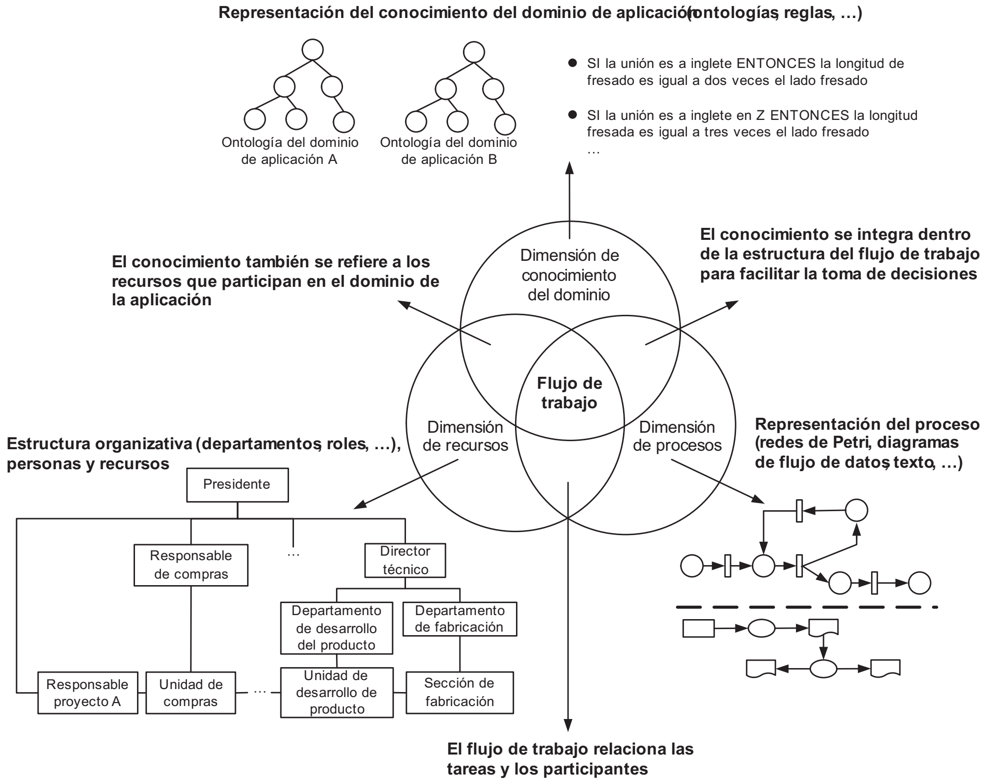


Figura 1.13: Metamodelo para la representación semántica de WFs en el que se introduce una nueva dimensión para el modelado del conocimiento en dominios complejos

La combinación de este metamodelo con un potente lenguaje de representación de WFs dota a estos sistemas de una gran capacidad de representación. También se han citado los principales problemas intrínsecos a este tipo de modelado, los cuales se podrían resumir en un único punto: *la falta de semántica del modelo* [130].

Con el objetivo de paliar este problema, en esta tesis doctoral consideramos la hipótesis de modelar los WFs desde un punto de vista semántico. Para ello se han tomado las principales propuestas para el modelado del conocimiento, analizando sus ventajas e inconveniente respecto a la representación de WFs. Las ventajas de este tipo de propuestas son las siguientes:

- *Arquitectura conceptual*. Los metamodelos analizados se estructuran pensando en la mejor forma de construir los procesos. Esta característica está también vinculada con la capacidad de reutilización de los componentes del modelo recogida en la Tabla 1.1. Por ejemplo, CommonKADS utiliza siete modelos donde cada uno de estos modelos captura un elemento básico del proceso, pero no indica cómo reutilizarlos. En cambio, metamodelos más recientes, como WSMO o UPML,

añaden conceptos como la *mediación* o la *adaptación* para facilitar la composición y reutilización de sus componentes. Por ejemplo, UPML establece tres adaptadores para combinar la descripción de los problemas a resolver (tareas) con una posible solución (método) en un determinado dominio de aplicación. De esta forma, cada componente del marco de conocimiento es independiente.

- *Integración del conocimiento.* Estos metamodelos están pensados para gestionar el conocimiento del proceso. Puede incluso afirmarse que en algunos de ellos, como UPML, únicamente se integra conocimiento. Cabe destacar que el conocimiento no está limitado a las reglas o axiomas de un determinado dominio de aplicación, sino que también se considera conocimiento a la estructura y características del proceso. En particular, los metamodelos analizados en este capítulo están enfocados a capturar este último tipo de conocimiento.

- *Razonamiento.* Varios de los metamodelos analizados describen los procesos a través de ontologías cuya representación sigue un paradigma lógico como la lógica descriptiva, la lógica basada en marcos o la lógica de primer orden. Facilitan así el razonamiento acerca de las características de los procesos y posibilitan que, por ejemplo, se puedan realizar consultas semánticas sobre las entradas, salidas o funcionalidad de los procesos.

Sin embargo, las propuestas centradas en el modelado del conocimiento tienen un importante inconveniente con respecto a los metamodelos de WFs tradicionales: *la representación del control.* Ninguna de estas propuestas, a excepción de WSMO, está basada en un formalismo de representación de procesos. Esto significa que la semántica de conceptos como la concurrencia o la sincronización no se contemplan ni se resuelven dentro de estos metamodelos. La Tabla 1.1 compara el marco tradicional de modelado de WFs con los distintos metamodelos de conocimientos y en ella se compruaba que la solución tradicional no es suficiente para reutilizar y procesar adecuadamente el conocimiento del WF. Esta tabla también nos permite apreciar los problemas de los marcos de conocimiento para capturar las características de los WFs. Así, únicamente CommonKADS contempla su modelo de organización. La mayoría de estos marcos tampoco gestionan correctamente la operacionalidad (es decir, la forma de invocar la operación a resolver) al no tener en cuenta que los recursos pueden ser los encargados de su resolución. WSMO y OWL-S resuelven parcialmente este problema si se consideran a los servicios como recursos/agentes del sistema. El modelado del comportamiento es otro de los puntos débiles de los marcos de conocimiento y se debe a que no son capaces de representar los patrones de comportamiento típicos de los WFs. En el caso de los componentes de la experiencia y de tareas genéricas el '-' se debe a que modelan el comportamiento a través de un lenguaje simbólico muy cercano al nivel de implementación. La conclusión es obvia: *para poder modelar WFs mediante marcos de conocimiento es necesario enriquecer su descripción con una semántica operacional* basada en un formalismo de representación de procesos.

La idea es que a través del metamodelo se facilite la estructuración, composición, y reutilización de los procesos y de paso poder responder a preguntas como ¿están ejecutándose dos procesos en paralelo?, ¿de qué otros procesos depende un proceso?,

Tabla 1.1: Ventajas y desventajas de las distintas aproximaciones para el modelado de WFs: 1-Marco tradicional de WFs, 2-CommonKADS, 3-Componentes de la experiencia, 4-Tareas genéricas, 5-Protégé II, 6-UPML, 7-WSMO, 8-OWL-S. Los signos '+' indican un soporte directo, los '+/-' un soporte parcial o incompleto y los '-' que no soporta la característica indicada. Es necesario precisar que un '-' no implica que sea imposible representar la característica indicada, sino que la solución no es directa o adecuada.

	1	2	3	4	5	6	7	8
Modelado de las características de los WFs								
Funcionalidad	-	+	+	+	+	+	+	+
Comportamiento	+	+/-	-	-	+/-	+/-	+/-	+
Tratamiento de la información	-	+/-	-	-	+	+	+	+
Operacionalidad	+	-	-	-	-	-	+/-	+/-
Organización	+	+	-	-	-	-	-	-
Reutilización del conocimiento								
Proceso	-	+/-	-	-	+	+	+	+
Dominio	-	+/-	-	-	+	+	+	+
Recursos	-	+/-	-	-	-	-	-	-
Razonamiento								
Modelo procesable	-	-	-	-	+	+	+	+

¿qué tareas deben finalizar para que un proceso continúe con la ejecución?, etc. Es conveniente precisar que para ello no bastaría con introducir una capa conceptual con las construcciones de control más comunes. Es decir, no sería suficiente una capa en la que se definiesen conceptos como la secuencia, la separación o la sincronización si no se dotase a dichos conceptos de una semántica operacional: esta capa tendría su semántica en las descripciones del texto asociado a los conceptos, lo cual no sería suficiente para razonar acerca las características del control. Para solucionar este problema, en esta tesis doctoral se ha optado por explicitar a través de una ontología uno de los formalismos de representación de WFs que se describirán en el capítulo 2 y utilizarlo para representar el control. Así, conseguimos, por una parte, formalizar la descripción de los procesos dentro del marco de conocimiento, y por otra, mejorar la capacidad de razonamiento acerca de las características estructurales del modelo de control.

Como consecuencia, un marco de conocimiento con estas características podría usarse para modelar WFs. Para ello, intentamos modelar semánticamente *cada una* de las componentes que intervienen en la definición de un WF, de forma que *(i)* se pueda razonar acerca de sus características y *(ii)* se puedan reutilizar de una manera más eficiente. Esta habilidad para gestionar el conocimiento sería una importante ayuda e innovación al desarrollo de WFs, ya que permitiría tratar cada componente de un WF como una pieza de conocimiento de forma independiente. Por ejemplo la Figura 1.13 muestra cómo quedaría el marco tradicional de WFs separando el modelo del dominio del proceso. El aspecto más destacado de la figura es que considera cada dimensión como una pieza de conocimiento que puede ser definida independientemente de las otras, y donde las intersecciones representan la adaptación entre componentes de distinto tipo. El nuevo modelo sigue manteniendo las dimensiones de procesos y de

recursos del marco tradicional de WFs y añade la dimensión de conocimiento del dominio. En esta nueva dimensión se modela el conocimiento a través de ontología y reglas, y participa activamente en la toma de decisiones que afectan a la ejecución y enrutamiento del WF (intersección con la dimensión de procesos). También facilita la selección de los recursos más adecuados para la ejecución de cada tarea (intersección con la dimensión de recursos).

Ante la necesidad de dotar de una mayor semántica operacional a los marcos conceptuales basados en conocimiento, en el siguiente capítulo se analizarán los principales lenguajes de representación WFs con el fin de determinar el lenguaje más indicado para nuestro propósito. Para ello se analizarán estos lenguajes en base a *(i)* su expresividad y *(ii)* su legibilidad.

2

Lenguajes de modelado de flujos de trabajo

En este capítulo se analizarán las aproximaciones al modelado de WFs desde tres perspectivas distintas: control, organización y conocimiento. Estos tres tópicos constituyen el área de interés de los WFs que hacen uso intensivo de conocimiento y, por lo tanto, de los procesos que esta tesis doctoral trata de modelar. El objetivo de este análisis es determinar el formalismo o lenguaje de representación más adecuado para capturar la semántica de estos procesos.

2.1. Características de los lenguajes de flujos de trabajo

Muchas han sido las propuestas de lenguajes de representación de WFs. La razón de tal disparidad de lenguajes y modelos se debe a la amplia utilización del concepto de WF, el cual puede enfocarse de distintas maneras en función de sus características, enfoque o paradigma de representación, como se ha descrito en el apartado 1.2.2. Tal disparidad hace que muchos lenguajes no sean directamente comparables, ya que sus características se basan en criterios subjetivos (por ejemplo en el entorno o contexto en que se desarrolla la aplicación) o en la herramienta utilizada para automatizarlos. Sin embargo, en los últimos años han aparecido pautas que permiten evaluar estos lenguajes mediante criterios más objetivos [153].

2.1.1. Expresividad del lenguaje

La expresividad da cuenta de la capacidad que tiene un lenguaje para representar un WF. Esta propiedad suele medirse en base a aspectos como el comportamiento, la operacionalidad o la organización. El problema de cómo medir la expresividad de un lenguaje ha sido muy discutido y tradicionalmente la forma genérica de comprobar que un lenguaje era más expresivo que otro consistía en plantear un conjunto de problemas y ver si dichos lenguajes eran capaces de resolverlos. El lenguaje más expresivo era aquél en el que la solución al problema se podía expresar con mayor claridad y menor complejidad. En este contexto, en 1999 se creó una iniciativa[1] entre la Technology

[1] http://www.workflowpatterns.com/

University of Eindhoven y la University of Technology Queensland con el objetivo de crear y mantener un conjunto de patrones que definan los aspectos esenciales que deben cumplir los WFs. En la actualidad, estos patrones se han convertido en la medida de expresividad de los modelos de representación, de modo que cuantos más patrones pueda representar un lenguaje, mayor será su expresividad. Además, esta iniciativa también actúa como observatorio y realiza una evaluación continua de las herramientas de gestión de WFs disponibles en el mercado.

En la actualidad existen cuatro categorías de patrones de WFs: los que tratan el *comportamiento*, los *datos*, los *recursos* y las *excepciones*. Sin embargo, debido a la fuerte influencia del modelado de procesos, los patrones de comportamiento son los más usados para medir la expresividad de los lenguajes de WFs, ya que con ellos se coordinan las actividades que, al fin y al cabo, son el núcleo del proceso. Los demás tipos de patrones *decoran* el proceso, y describen situaciones más dependientes de la implementación que del proceso en sí.

2.1.1.1. Patrones de comportamiento

Estos patrones, también llamados *patrones de control*, se corresponden con la revisión de los patrones presentados en [221, 260]. Su principal objetivo es identificar las distintas formas de coordinar las tareas que componen un WF. Dentro de esta categoría destacan los siguientes patrones:

Control básico. Modelan estructuras básicas de control que incorporan la mayoría de los lenguajes de representación. En total se definen cinco estructuras básicas que permiten modelar la *secuencia* (*sequence*, en inglés), la *separación* (*parallel split*, en inglés), la *sincronización* (*synchronization*, en inglés), la *elección exclusiva* (*exclusive choice*, en inglés), y la *mezcla* (*merge*, en inglés). La Figura 2.1 muestra dichas construcciones donde la parte de la derecha de la figura muestra el estado del patrón después de ejecutarse la construcción de control (el estado se representa mediante una marca negra). Así:

- Una *secuencia* establece el orden de ejecución de un conjunto de tareas, es decir, una tarea no podrá ejecutarse hasta que todas las tareas con mayor orden hayan finalizado su ejecución. Por ejemplo, en la secuencia representada en la Figura 2.1 se puede ver cómo la ejecución de la tarea A activa a la tarea B para ser ejecutada y, por lo tanto la marca negra se posiciona en la tarea B.
- Una *separación* (*parallel split* o *AND-split*, en inglés) parte un hilo de ejecución en varios (sub)hilos paralelos. Por ejemplo, en la separación representada en la Figura 2.1 se puede observar como la ejecución del nodo *AND*, que representa el punto de separación, elimina la marca de la tarea A y genera dos marcas en las tareas de salida B y C. Estas dos tareas no están relacionadas y, por lo tanto, pueden ejecutarse de forma independiente.
- Una *sincronización* (*synchronization* o *AND-join*, en inglés) une varios (sub)-hilos de ejecución en un único hilo. Por ejemplo, se puede observar en la Figura

2.1 como el nodo *AND*, que representa el punto de sincronización, une los dos hilos de ejecución de las tareas de entrada A y B en el hilo de la tarea de salida C. Para que la sincronización tenga efecto los hilos concurrentes han de estar activos al mismo tiempo, lo que implica que tanto la tarea A como la tarea B deben tener una marca para que se ejecute el nodo *AND*.

- Una *elección exclusiva* (*exclusive choice*, *XOR-split* o *exclusive OR-split*, en inglés) tiene la misma estructura que el patrón *separación* pero un comportamiento diferente. En este caso el nodo *XOR* representado en la Figura 2.1 permite seleccionar cuál de las dos ramas de salida activar, es decir, si activar la tarea B (caso a) o la tarea C (caso b).

- Una *mezcla simple* (*simple merge XOR-join* o *exclusive OR-join*, en inglés) se estructura como el patrón *sincronización*, pero no necesita que todas las tareas antecedentes estén en este momento en ejecución para activar la tarea de salida. Por ejemplo, en la Figura 2.1 el nodo *XOR* puede activarse siempre y cuando la tarea A, la tarea B o ambas tareas estén ejecutándose.

Control avanzado. Modelan estructuras más avanzadas que pueden llegar a surgir en un proceso de negocio. Estos patrones se caracterizan por tener unas estructuras más complejas de ramificación (como la separación múltiple o la elección múltiple) o de sincronización (como la mezcla múltiple o la discriminación). Este tipo de patrones no son muy comunes en la práctica y por ello no están soportados por muchos de los sistemas comerciales [260]. La Figura 2.2 representa la estructura y comportamiento de cuatro de los catorce patrones de control avanzado más representativos:

- Una *elección múltiple* (*multi-choice*, en inglés). Extiende el patrón básico de *elección exclusiva*, de forma que más de una rama de salida pueda ser seleccionada. Por ello en la figura aparecen las opciones a, b y c habilitadas.

- Una *mezcla múltiple* (*multi-merge*, en inglés), que extiende el patrón de *mezcla simple* y genera una salida para cada entrada.

- Una *unión parcial* (*structured partial join*, en inglés) donde únicamente se requiere la finalización de un determinado número de ramas para finalizar la estructura de control. En el ejemplo es suficiente con que dos de las tres ramas hayan finalizado su ejecución para que se ejecute la salida.

- Finalmente, una *discriminación* (*structured discriminator*, en inglés) permite unir un conjunto de ramas paralelas, aunque dicha unión no tiene por qué ejecutarse de forma sincronizada. En el ejemplo de la figura se puede apreciar cómo en un primer paso se ejecuta la tarea B y en un segundo paso la tarea A.

Los demás patrones de esta categoría definen variaciones de los patrones de sincronización, mezcla y discriminación.

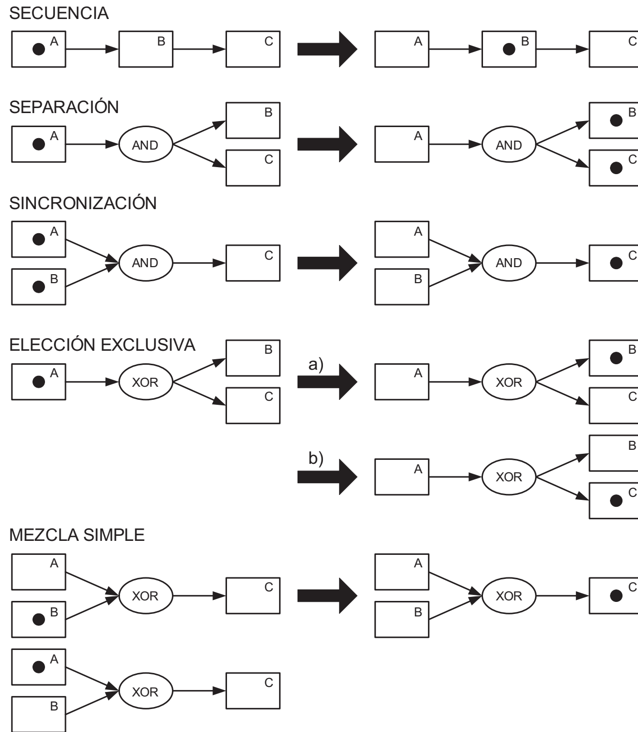


Figura 2.1: Patrones básicos de control. Cada rectángulo representa una actividad, las elipses representan construcciones de control y los arcos dirigidos representan las dependencias entre los distintos elementos.

Instanciación múltiple. Modelan situaciones donde puede existir más de un hilo de ejecución activo para una misma actividad y donde las distintas instancias comparten la misma implementación. La instanciación múltiple puede darse en tres situaciones:

1. Una actividad puede iniciar múltiples instancias de sí misma. Aunque estas instancias son independientes entre sí y, por lo tanto, pueden ejecutarse concurrentemente, cada una de ellas debe ejecutarse dentro del contexto de la instancia del proceso que las generó (por ejemplo, compartiendo el mismo identificador de proceso/caso y los mismos datos).

2. Una actividad puede ser iniciada múltiples veces. Por ejemplo si la actividad está en un bucle o si existen varias ramas concurrentes que inician la misma actividad.

3. Dos o más actividades en un proceso comparten la misma implementación. En WFs administrativos suele ser habitual que una misma tarea, o por lo menos una tarea muy similar, se repita a lo largo de la estructura. Aunque dichas tareas

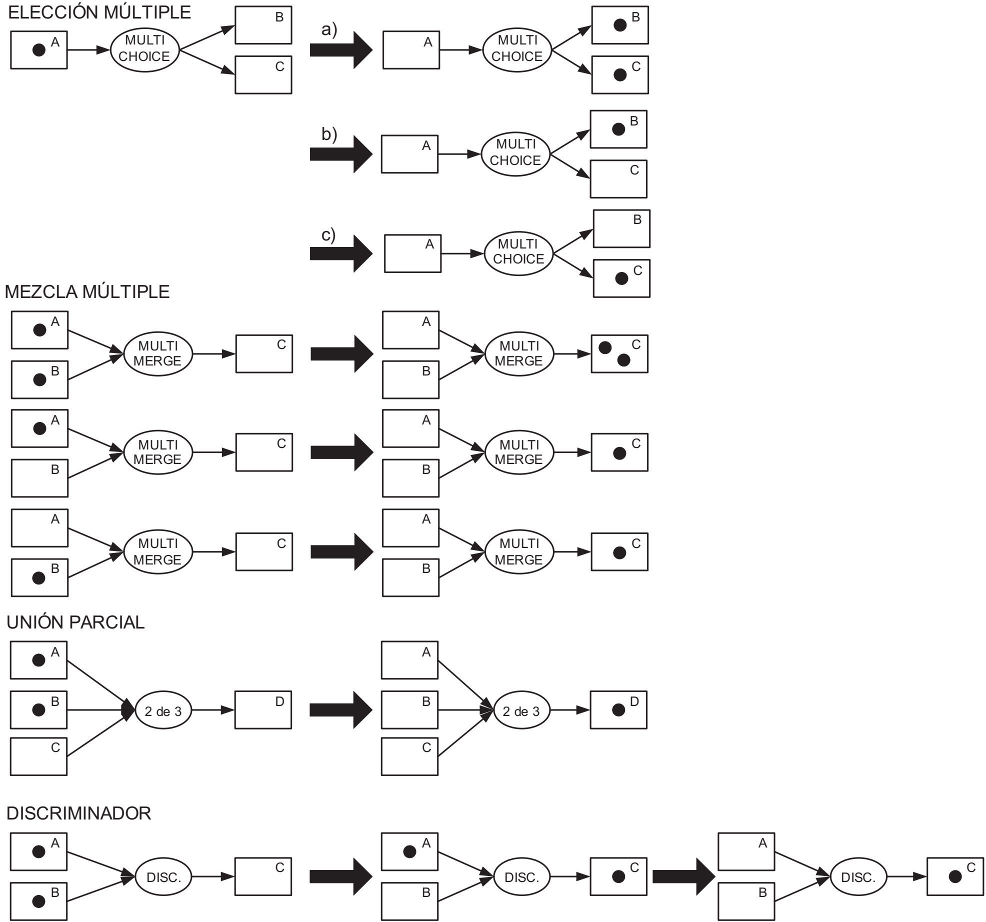


Figura 2.2: Cuatro patrones avanzados de control

sean distintas, bien por sus características bien por su localización dentro del WF, podrían tener la misma implementación.

Sin embargo, estos patrones únicamente se refieren a la primera situación. Este grupo de patrones está enfocado a las distintas formas en las que se pueden separar y sincronizar múltiples instancias. Por ejemplo, la Figura 2.3 muestra la estructura del patrón *unión parcial para instanciación múltiple* (*static partial join for multiple instances*, en inglés), que trata la ejecución de la actividad con instanciación múltiple como un patrón de control avanzado, de forma que la actividad finaliza cuando dos de sus tres instancias hayan concluido.

Control basado en el estado. Algunas situaciones, especialmente aquellas que requieren tener una visión completa de la ejecución del WF, se pueden resolver más

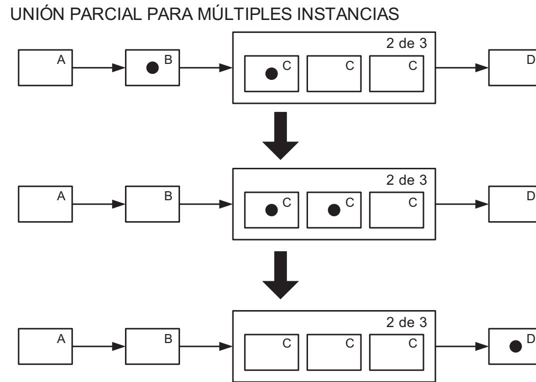


Figura 2.3: Ejemplo de un patrón para instanciación múltiple

fácilmente si el lenguaje permite representar el estado. En este contexto, se considera *estado del WF* a la instancia del proceso que debería incluir tanto el estado de todas las actividades como a los datos más relevantes del caso. Esta categoría incluye siete patrones en los que el estado actual determina la acción a realizar desde la perspectiva del flujo de control. La Figura 2.4 muestra la estructura de estos patrones:

- Los dos primeros ejemplos tratan el disparo y unión de un conjunto de hilos de ejecución. Específicamente, el primer patrón, llamado *separación hilo de ejecución*, permite crear $n = 3$ hilos de ejecución mientras que el segundo, llamado *mezcla hilo de ejecución*, permite sincronizar $n = 2$ hilos de ejecución.

- El tercer patrón se denomina *elección diferida* (*deferred choice*, en inglés) y modela el caso donde la decisión acerca de qué rama ejecutar se basa en la interacción con el entorno operativo del proceso. Al contrario de la *elección exclusiva* (control básico) no existe un criterio explícito de selección, sino una carrera entre las ramas para ver cuál será elegida.

- El patrón *enrutamiento paralelo alternado* (*interleaved parallel routing*, en inglés) modela una situación donde un conjunto de actividades pueden ejecutarse en un determinado orden parcial pero no al mismo tiempo.

- El patrón *enrutamiento alternado* (*interleaved routing*, en inglés) elimina la ordenación entre las actividades del patrón anterior, de forma que éstas puedan ejecutarse en cualquier orden.

- El patrón *hito* (*milestone*, en inglés) modela una situación donde una tarea estará activa únicamente cuando el WF se encuentre en un determinado estado, que se representa explícitamente en el modelo y se conoce con el nombre de *hito*. En la Figura 2.4, ese estado está modelado con el círculo que tiene el trazo de guiones.

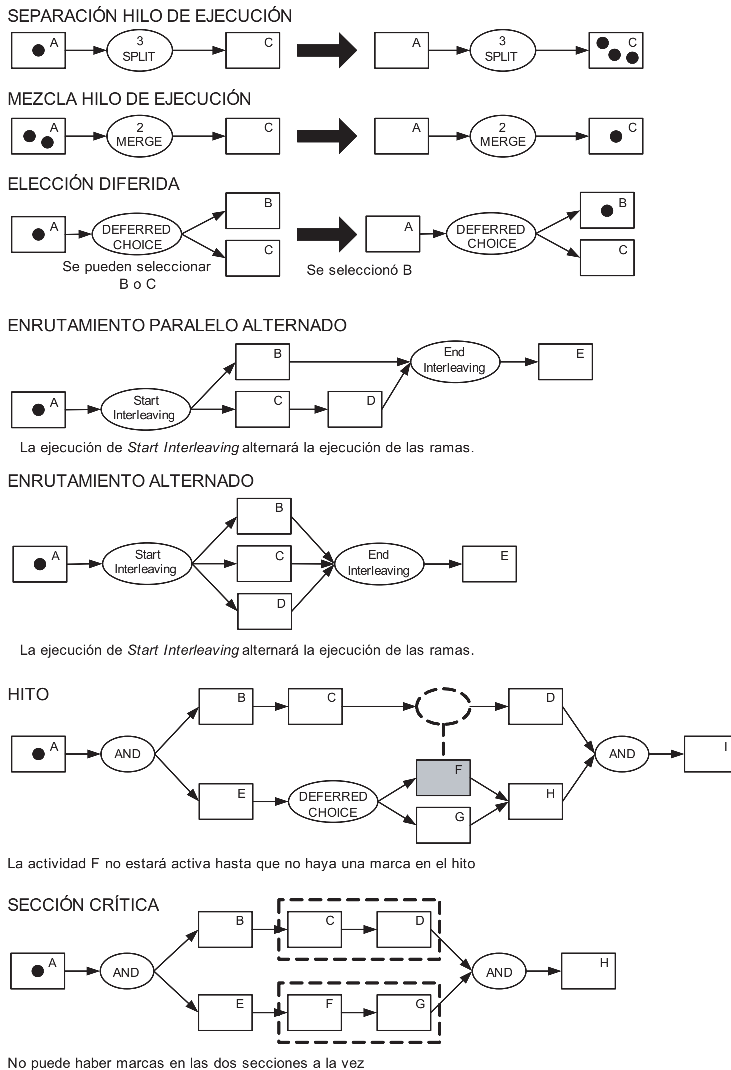


Figura 2.4: Patrones de control basados en el estado

- Finalmente, el patrón *sección crítica* (*critical section*, en inglés), permite identificar (sub)grafos del WF como secciones críticas, de forma que podrá estar activa a la vez una única sección crítica.

Cancelación. Esta categoría de patrones se utiliza para cancelar la ejecución de una parte o del proceso completo. La Figura 2.5 muestra algunos de los patrones de cancelación:

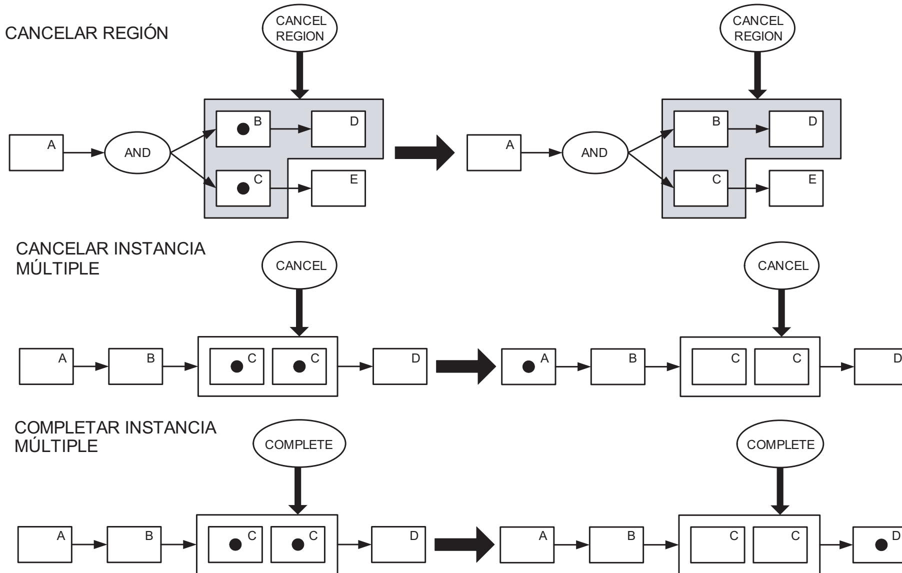


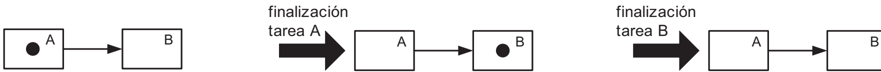
Figura 2.5: Ejemplo de tres patrones de cancelación

- El patrón *cancelar región* (*region cancellation*, en inglés) permite cancelar la ejecución de una determinada región del WF. Se puede observar en la figura cómo la cancelación de la región elimina todas las instancias de ejecución del proceso.

- El patrón *cancelar instancia múltiple* (*multiple instance cancellation*, en inglés) permite eliminar las instancias que se están ejecutando dentro de una tarea de instanciación múltiple.

- Finalmente, el patrón *completar instancia múltiple* (*multiple instance completion*, en inglés) muestra una alternativa al patrón anterior en la que se concluye la tarea de instanciación múltiple posteriormente a la eliminación de las instancias.

Terminación. Las redes representadas en la Figura 2.6 muestran las dos únicas opciones mediante las cuales se puede escenificar la finalización de un WF. La primera asume la terminación implícita: la instancia de un proceso debe finalizar cuando se han completado todas las actividades, es decir, cuando no se pueden realizar más actividades ni ahora ni en el futuro, y la instancia del proceso no está en un ciclo mortal. En el ejemplo de la figura se puede ver cómo tras la ejecución de las tareas A y B no se aprecia ninguna marca de ejecución en la red de la derecha, con lo cual se considera la ejecución finalizada. El segundo patrón asume la terminación explícita, y modela expresamente el final del WF a través de un nodo de terminación, de modo que el proceso habrá finalizado cuando se haya alcanzado ese nodo. Esta finalización



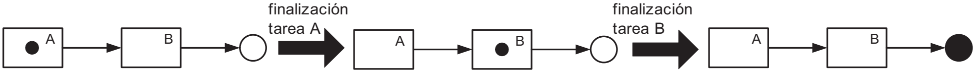


Figura 2.6: Patrones de terminación

implica la cancelación de todo el trabajo relacionado con la instancia del proceso independientemente de si alguna de las tareas aún está en ejecución. En la Figura 2.6 se modela esta clase de terminación con un círculo.

Iterativo. Estos patrones permiten modelar estructuras iterativas. Específicamente, se definen tres clases de bucles: estructurados, arbitrarios y recursivos. Los bucles estructurados son los más conocidos, ya que representan estructuras típicas de programación. Esta clase incluye entre otros a los bucles tipo *repetir-mientras* o *repetir-hasta*. La segunda clase, denominada *bucles arbitrarios*, representa estructuras iterativas donde el bucle puede tener más de un punto de entrada y más de un punto de salida. No tiene una correspondencia directa con ninguna instrucción de programación pero existe una cierta similitud entre este patrón y la combinación de instrucciones tipo *goto* utilizadas en programación no estructurada. Finalmente, la última clase representa la habilidad de una actividad de (auto)invocarse o de invocar a un antecesor en términos estructurales. La Figura 2.7 representa gráficamente los distintos tipos de patrones.

Control basado en eventos. El inicio de una actividad puede requerir la intervención de una señal/evento externo. Los dos tipos de patrones que pertenecen a esta categoría se distinguen si el disparador de la señal es transitorio o permanente. La Figura 2.8 representa estos dos patrones. En la parte superior representa un caso de uso del patrón *evento externo transitorio* (*transient trigger*, en inglés). Se observa en la parte izquierda del ejemplo que la ocurrencia del evento no se almacena y al no estar activa la tarea C, el evento se pierde. En cambio, se puede advertir en el ejemplo del patrón *evento externo transitorio* (*persistent trigger*, en inglés) cómo el WF mantiene el evento incluso no estando activa la tarea C. Así, cuando finalmente se ejecute la tarea C, el evento podrá aplicarse a la tarea.

2.1.1.2. Patrones de datos

Estos patrones modelan la utilización de los datos en los WFs y se corresponden con una revisión de los patrones presentados en [219, 216]. Dentro del contexto de un proceso, los datos pueden definirse y utilizarse de muchas formas, y por ello es necesario modelar aspectos como:

Visibilidad. Típicamente, la visibilidad de unos datos suele estar restringida a una

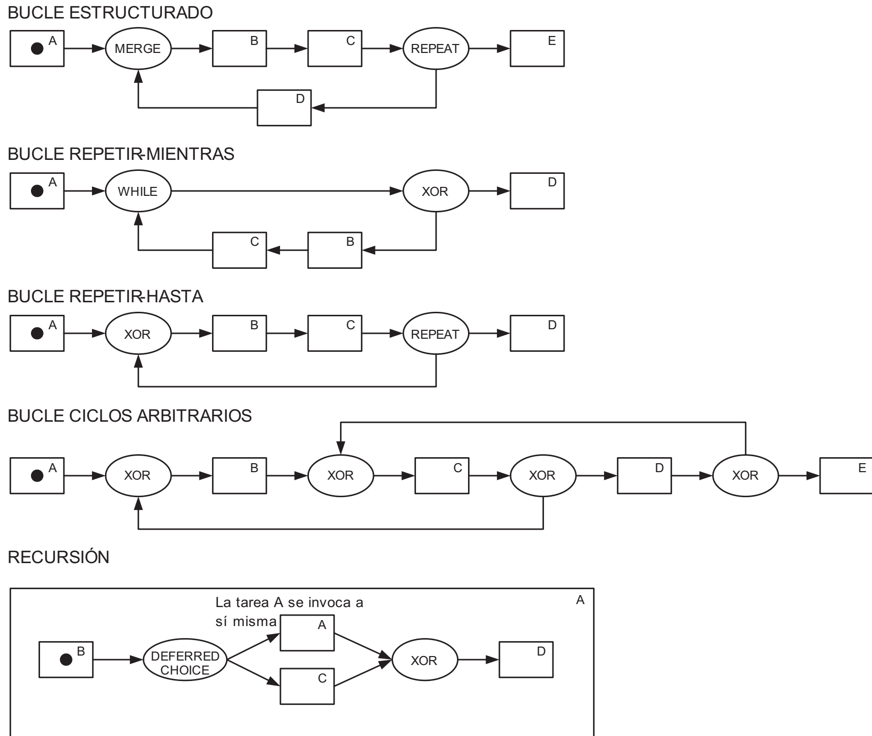


Figura 2.7: Patrones que modelan estructuras iterativas

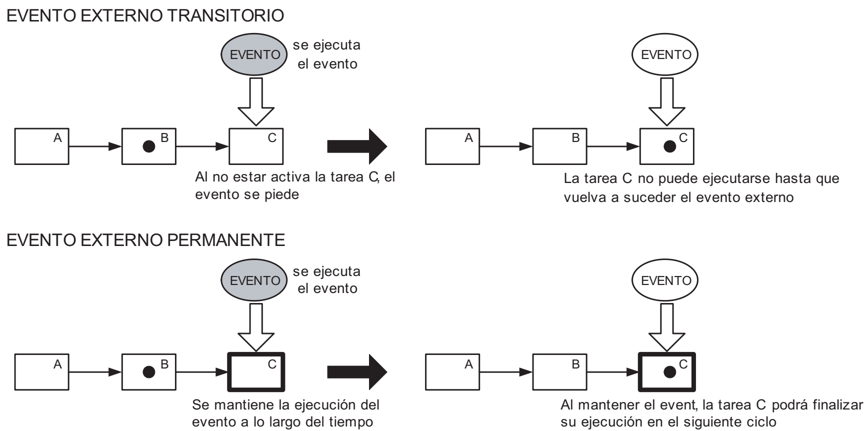


Figura 2.8: Patrones de control basado en eventos

tarea, un bloque, un caso o al propio WF. La Figura 2.9 muestra estos cuatro principales tipos de visibilidad. La zona coloreada en gris circunscribe la zona en la que las

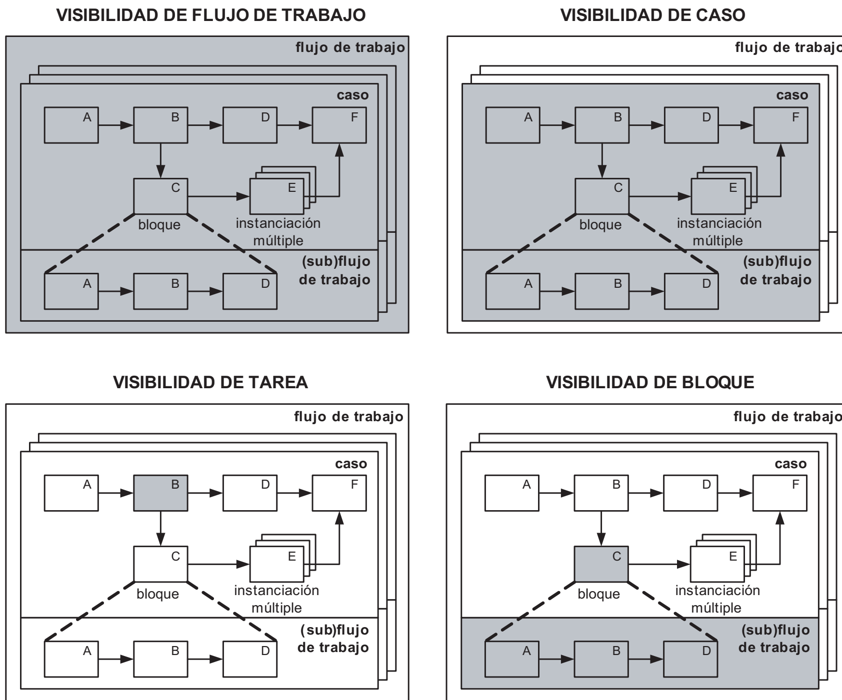


Figura 2.9: Patrones de visibilidad

variables están visibles. Los datos con visibilidad de WF son accesibles desde cualquier caso (o instancia de ejecución), tarea, construcción de control o (sub)flujo. Cuando la visibilidad se restringe al caso, los datos serán accesibles por parte de todas las tareas, construcciones de control y (sub)flujos que estén participando en la ejecución del caso. Si los datos tienen visibilidad de tarea, sólo serán accesibles dentro de la tarea. Finalmente, la visibilidad de bloque se refiere al acceso a los datos de una tarea cuando ésta se resuelve mediante un (sub)flujo.

La accesibilidad a los datos no se restringe a estos elementos y se definen patrones de visibilidad para otros elementos herencia de los sistemas tradicionales de WFs:

- *Ámbitos*. Comparten datos entre un conjunto de tareas
- *Carpetas*. Comparten datos entre distintos casos
- *Entornos*. Se refieren a los datos disponibles en repositorios, servicios y aplicaciones externas que podrán ser accedidos por los elementos del WF.

Interacción. Los patrones de interacción tratan las distintas formas en las que se pueden pasar datos entre distintos componentes de un proceso y cómo las características de los componentes pueden llegar a influenciar la forma en la que se produce el

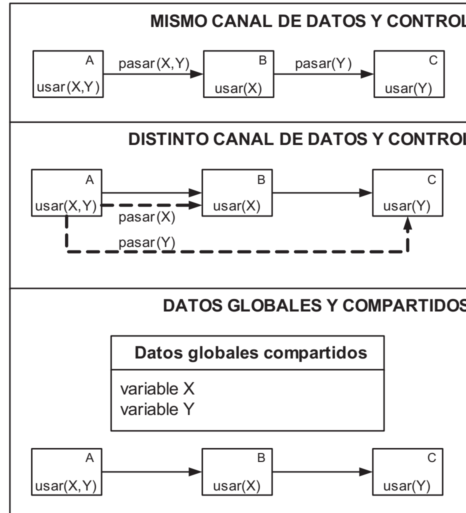


Figura 2.10: Patrón de interacción entre dos tareas

tránsito de la información. Se pueden distinguir dos tipos de patrones de interacción. El primer tipo trata la comunicación entre componentes de un mismo proceso, por ejemplo, entre dos tareas, dos casos o entre un bloque y un (sub)WF. Por ejemplo, la Figura 2.10 muestra las tres posibles formas de comunicación entre dos tareas: compartiendo un mismo canal (o bus de comunicaciones), compartiendo un canal para datos y otro canal para el control, y compartiendo la misma zona de datos. El segundo tipo de patrones trata la interacción entre un componente de un proceso y un entorno externo, por ejemplo, entre una tarea y el entorno, entre el WF y el entorno, y viceversa. Básicamente se definen dos tipos de interacción, subir y bajar (*push* y *pull*, en inglés) , dependiendo del elemento que inicie la comunicación.

Transferencia. Los patrones de transferencia tratan la forma en la que se pasan los datos entre componentes de un proceso. El patrón *transferencia de entrada por valor* permite pasar los datos de entrada de una tarea por valor evitando la necesidad de tener nombres compartidos o espacios de direcciones comunes entre el componente que manda y el componente que recibe. De la misma forma, el patrón *transferencia de salida por valor* permite el pase de datos por valor al siguiente elemento del WF. Por el contrario, los patrones *transferencia por referencia con bloqueo* y *transferencia por referencia sin bloqueo* tratan el pase de parámetros por referencia, es decir, mediante el pase de una referencia a la localización del dato en algún dispositivo accesible por el emisor y el receptor. Ambos patrones se diferencian en las restricciones de concurrencia establecidas respecto a los datos compartidos. Así, el primer patrón establece un acceso dedicado al receptor de forma que los demás elementos del WF que accedan a ese mismo dato tengan acceso de sólo lectura. Otro patrón, denominado *transferencia por copia* permite *(i)* copiar los valores de un conjunto de datos desde una fuente externa a un espacio de direcciones y al finalizar la ejecución de la tarea

(ii) volver a copiar los datos con sus valores finales nuevamente a la fuente externa. Finalmente, los dos últimos patrones de esta categoría se refieren a la transformación de los datos justo antes de la transferencia más que al propio proceso de transferencia. Por un lado, el patrón *transformación de entrada* se refiere a la transformación de los datos justo antes de ser pasados a la tarea. Por el otro, el patrón *transformación de salida* se refiere a la transformación de los datos justo antes de ser pasados al siguiente elemento del WF.

Enrutamiento basado en datos. Los patrones de enrutamiento describen la influencia de los datos en la toma de decisiones de control. Varios de los patrones de esta categoría tratan los tipos de precondiciones y postcondiciones más utilizados en los WFs: el patrón *precondición - existencia dato* verifica la disponibilidad de algún dato antes de la ejecución de la tarea; el patrón precondición - valor dato verifica si un dato tiene un determinado valor en tiempo de ejecución; los patrones *postcondición - existencia dato* y *postcondición - valor dato* tienen el mismo significado que en el caso de las precondiciones pero en este caso aplicado al completarse la tarea. Otros dos patrones de esta categoría tratan la forma de disparar/iniciar las tareas. El primero de ellos, denominado *disparo de tarea por evento* se refiere al comienzo de la tarea a partir de un evento externo y de los datos que ese evento le proporciona a la tarea. El segundo, denominado *disparo de tarea por dato*, se refiere al comienzo de la tarea cuando una determinada expresión basada en los datos de la instancia del proceso se evalúa a verdadero. Finalmente, el último patrón de esta categoría, denominado *enrutamiento basado en datos* alude al enrutamiento de los elementos de control de los WFs. Este patrón se refiere a las expresiones que algunos elementos de control pueden utilizar para tomar sus decisiones. Por ejemplo, los criterios de selección de un patrón de control básico como la *elección exclusiva* o el patrón avanzado *conflicto* (*or split*, en inglés).

2.1.1.3. Patrones de organización

Estos patrones, también llamados patrones de recursos [225, 217], están centrados en el modelado de los recursos, independientemente de si son humanos o software, y en su interacción con los sistemas de gestión de procesos de negocio. Esta categoría está compuesta por los siguientes tipos de patrones:

Creación. En este tipo de patrones se definen limitaciones en la forma en la cual se puede ejecutar un trabajo. Estas limitaciones se refieren, por un lado, al tipo de recurso que puede realizar la tarea y, por el otro, a los criterios de distribución del trabajo. En términos del ciclo de vida de un trabajo, los patrones de creación se ejecutan en el momento en el que se crea el trabajo. Por ejemplo, la Figura 2.11 muestra los estados por los que tiene que pasar el trabajo. Como se puede apreciar, la fase de creación es la primera en ejecutarse. La ejecución de la actividad *CREAR* ejecutará un determinado patrón que definirá el modelo organizativo en el que se identificarán unívocamente los recursos y el mecanismo de distribución del trabajo entre los recursos identificados en el modelo organizativo. Estos patrones se suelen especificar en tiempo de diseño, donde se define el modelo de distribución a la vez que

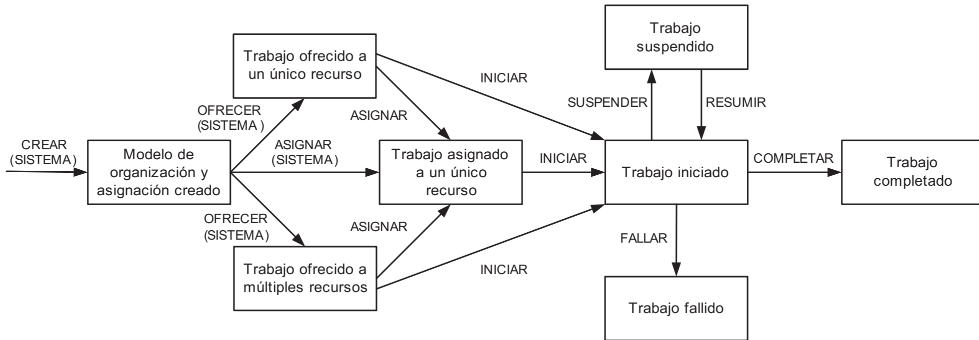


Figura 2.11: Ciclo de vida del trabajo

el modelo organizativo. Existen muchos criterios de distribución del trabajo, aunque los más utilizados son:

- *Directa.* Asignación del trabajo a las instancias de un determinado recurso.
- *Basada en roles.* Asignación del trabajo a las instancias que tengan un determinado rol.
- *Diferida.* Asignación del trabajo a las instancias de un determinado recurso. El recurso al que asignar el trabajo se determinará en tiempo de ejecución.
- *Autorización.* La asignación de permisos a los recursos define qué trabajos pueden realizar.
- *Separación de responsabilidades.* Define un criterio de asignación a nivel de caso. El criterio especifica qué dos tareas deben ser ejecutadas por dos recursos distintos.
- *Manejo de casos.* Asignación del trabajo referente a un caso a un determinado recurso.
- *Casos familiares.* Asignación del trabajo al recurso que trató un caso similar.
- *Habilidad.* Asignación del trabajo basado en las habilidades de los recursos. Las habilidades deben estar asociadas al recurso en el modelo de la organización.
- *Historial.* Asignación del trabajo basado en el historial de ejecución.
- *Organización.* Asignación del trabajo basado en la posición del recurso en la organización y en sus relaciones con otros recursos.
- *Ejecución automática.* Ejecución de la tarea sin necesidad de utilizar recursos.

Ofrecimiento (*push pattern*, en inglés). En este tipo de patrones modela el ofrecimiento de trabajo a los recursos por parte del sistema. El sistema puede *(i)* ofrecer trabajo a un único recurso, *(ii)* ofrecer el trabajo a un grupo de recursos o *(iii)* directamente asignar el trabajo a un recurso. Estos patrones definen los criterios de selección del recurso al que ofrecer o asignar el trabajo: directa, asignación aleatoria, round robin, basada en colas de trabajo, distribución antes de activarse el trabajo, distribución al activarse el trabajo, distribución después de activarse el trabajo.

Asignación (*pull pattern*, en inglés). En este tipo de patrones modela una asignación del trabajo guiada por el propio usuario. Una vez ofrecido un trabajo aparecerá en la lista de trabajos del recurso y éste podrá (auto)asignarse o iniciar su ejecución. Si se observa la Figura 2.11, estos patrones tratan las tareas de asignación e inicio del trabajo que no están soportadas por el sistema.

Rodeo (*detour*, en inglés). Otro tipo de patrones tratan la interrupción del trabajo, bien sea ésta motivada por una acción del sistema o por un usuario. Como consecuencia de este tipo de evento, la asignación existente previa a la interrupción suele variar y estos patrones definen las nuevas formas de asignar el trabajo. En estas situaciones las estrategias más comunes permiten delegar, escalar o reasignar el trabajo a otro recurso.

Inicio automático. Los patrones de inicio automático se refieren a situaciones donde la ejecución del trabajo está determinada por eventos específicos bien definidos en el ciclo de vida del trabajo o en la propia definición del proceso. Estos eventos permiten la creación, asignación o la finalización de un trabajo.

Visibilidad. Estos patrones tratan la visibilidad del trabajo respecto a determinados recursos en función de su responsabilidad. La visibilidad afecta a los criterios de ofrecimiento, asignación e inicio del trabajo.

Multiples recursos. Estos patrones tratan la ejecución de un trabajo por medio de un grupo de recursos.

2.1.1.4. Patrones de gestión de excepciones

Estos patrones definen soluciones para el manejo de las excepciones durante el la ejecución del WF. Estos patrones, presentados en [223, 222], analizan el tratamiento de los distintos tipos de excepciones, el nivel donde las excepciones deben tratarse, la recuperación del WF en caso de una excepción y las distintas estrategias de tratar una excepción. Por ejemplo, la Figura 2.12 muestra los estados en los que puede estar un trabajo. Las líneas con trazo continuo representan el camino normal del trabajo: un trabajo es ofrecido, asignado y se completa o falla. Las líneas discontinuas muestran aquellas situaciones donde el comportamiento se sale de lo normal y es necesario tomar medidas para reconducirlo.

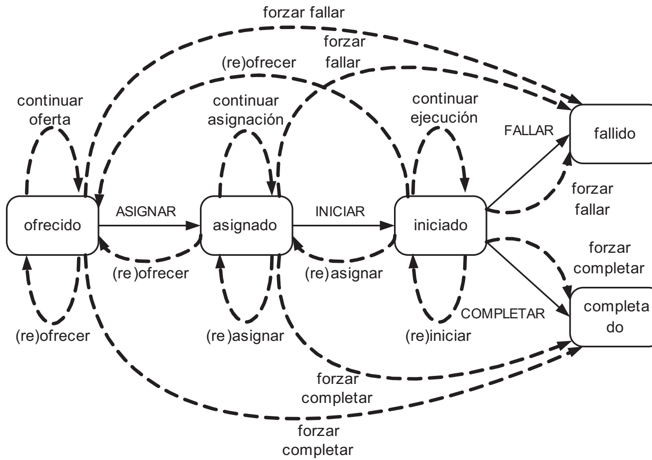


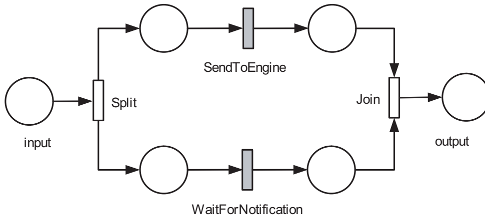
Figura 2.12: Estados del trabajo

2.1.2. Legibilidad del modelo

La legibilidad es otro de los factores determinantes a la hora de seleccionar un lenguaje de representación de WFs. En este contexto, se entiende por legibilidad a la facilidad con la que un diseñador puede entender un determinado modelo. Sin embargo, es un concepto muy subjetivo y lo que puede parecer legible para un programador informático puede no serlo tanto para un experto ajeno a este ámbito. Por ello, en función de la experiencia del diseñador, criterios como la legibilidad pueden primar incluso más que la expresividad del lenguaje.

Un aspecto necesario para que un lenguaje sea legible es que sea gráfico. Por ejemplo, la Figura 2.13 muestra dos modelos de WF especificados en dos lenguajes distintos. El modelo de la izquierda está basado en el formalismo gráfico de las redes de Petri mientras que el de la derecha en el lenguaje BPEL. Está claro que una persona sin experiencia en informática tendría menos problemas en asimilar el modelo gráfico que el modelo textual de BPEL. Está demostrado que el uso de lenguajes gráficos facilita la comprensión de los modelos de WFs, y por ello la mayor parte de los sistemas comerciales de WFs suelen incorporar algún tipo de representación gráfica. Estos sistemas están pensados para ser usados por personas que no tienen conocimientos de programación, de forma que los propios expertos del dominio puedan diseñar la estructura del proceso de negocio.

Otro factor que afecta la legibilidad es la visualización del modelo: si un modelo se puede diseñar en varios módulos la inteligibilidad del modelo completo será mayor. Este aspecto también está relacionado con el tamaño del modelo. Cuantos menos elementos utilice el modelo, más sencilla será su interpretación. Por ejemplo, la Figura 2.14 muestra un mismo modelo representado a través de dos tipos distintos de redes de Petri. Mientras el modelo de la izquierda usa redes de bajo nivel, el de la derecha utiliza redes de alto nivel y jerarquizadas. Aunque el ejemplo es sencillo, puede apreciarse




```
<process name ="WorkflowExample" >
  …
  <flow name ="Split" >
    <sequence name ="Send" >
      <invoke name ="SendToEngine"
          operation ="startComputation"
          inputVariable ="input"
          outputVariable ="output" />
    </sequence >
    <sequence name ="Wait" >
      <receiveNotification
          name="WaitForNotification"
          serviceData ="ComputationComplete"  />
    </sequence >
  </flow>
  …
</process >
```

Figura 2.13: Dos lenguajes de representación de WFs

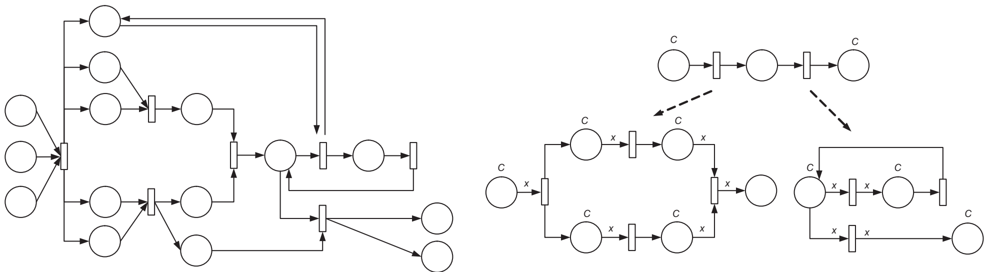


Figura 2.14: Efecto de la visualización a la hora de interpretar dos modelos

el incremento significativo de elementos en el caso de las redes de bajo nivel y cómo ello dificulta la interpretación del modelo completo. Asimismo, el no poder partir el modelo de la izquierda en (sub)elementos tampoco ayuda a su interpretación.

2.1.3. Tipos de lenguajes

En los siguientes apartados estudiaremos la legibilidad y expresividad de los principales lenguajes de WFs. Estos lenguajes pueden clasificarse en dos categorías atendiendo a su legibilidad:

- *Lenguajes de representación.* Son lenguajes visuales con una importante legibilidad y expresividad. Su problema es la falta de una semántica formal que permita trasladar sus diagramas a un modelo ejecutable.

- *Lenguajes de ejecución.* Son lenguajes basados en texto con menor legibilidad y expresividad que los basados en representación. A pesar de ello estos lenguajes son muy usados, ya que permiten la ejecución de WFs.

En la práctica, los WMSs suelen combinar un subconjunto de la expresividad de un lenguaje de representación con un lenguaje de ejecución y definir la adaptación entre la semántica de ambos lenguajes.

También incluiremos en este estudio un breve análisis de los principales *formalismos de representación* de procesos aplicados al modelado de WFs. Éstos, además de proporcionar un modelo matemático que asegura el correcto funcionamiento del modelo, disponen de técnicas de análisis y simulación. Estas utilidades son indispensables a medida que la complejidad del modelo crece, ya que permiten comprobar que el modelo construido es correcto.

2.2. Formalismos de representación de los lenguajes de flujos de trabajo

Los métodos formales aplicados al modelado de WFs son básicamente representaciones matemáticas de sistemas discretos que fueron desarrolladas para tratar la concurrencia entre procesos. En la práctica son a la concurrencia lo que los métodos clásicos, como la máquina de Turing [250] o el Lambda cálculo [23], son a la computación secuencial y funcional.

En la bibliografía se pueden encontrar muchos ejemplos de métodos formales aplicados al modelado de WFs. La gran mayoría están basados en el álgebra de procesos y en la teoría de autómatas. Existen algunas aproximaciones basadas en lógica (por ejemplo, lógica temporal o lógica de transacciones) [229, 82], pero pocos sistemas comerciales de WFs emplean este formalismo debido a los problemas para tratar adecuadamente la concurrencia. Por ello, este análisis se centrará en las tres principales aportaciones: las *redes de Petri*, el π-calculus y las *máquinas de estados abstractos*.

2.2.1. Redes de Petri

Las redes de Petri son un formalismo para modelar sistemas dinámicos discretos. Son gráficas, están formalizadas matemáticamente y se pueden desarrollar algoritmos para analizar las propiedades. Las redes originales fueron desarrolladas por Carl Adam Petri [203] y en la actualidad existen incontables extensiones. Su principal contribución es su gran expresividad, la suficiente para representar, en su versión de alto nivel, los patrones de comportamiento de los WFs [263]. Además, las redes de Petri pueden modelar sincronización de procesos, eventos asíncronos, operaciones concurrentes y compartición de recursos. Es por ello que estas redes son el marco formal más ampliamente usado para la formalización y simulación de procesos muy complejos. Una red de Petri clásica se define como un grafo dirigido bipartito:

$$PN = (P, T, A)$$

donde T es un conjunto finito de transiciones, P es un conjunto finito de plazas y A es un conjunto finito de arcos dirigidos entre plazas y transiciones y viceversa. Una de

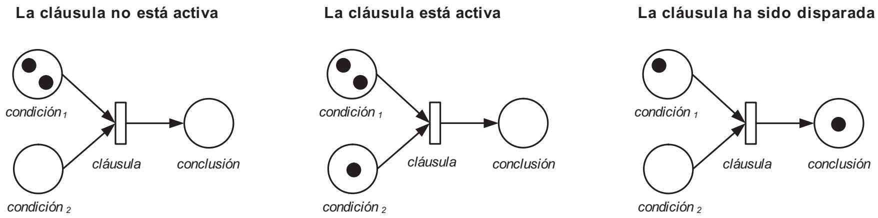


Figura 2.15: Ejemplo de marcado, activación y disparo de una transición donde los círculos representan a las plazas, los rectángulos a las transiciones y las marcas se representan mediante puntos negros dentro de las plazas

Tabla 2.1: Posibles interpretaciones de plazas y transiciones en redes de Petri

plazas de entrada	**transición**	**plazas de salida**
precondiciones	evento	postcondiciones
datos de entrada	paso de computación	datos de salida
recurso requerido	operación	recurso liberado
condiciones	cláusula lógica	conclusiones

las mayores utilidades de estas redes es su capacidad para representar dinámicamente la evolución de los sistemas. Para este propósito, el formalismo de las redes de Petri se define como una 4-tupla:

$$PN = (P, T, A, M)$$

donde M es una función de marcado asociada a cada una de las plazas de la red. Así, las plazas permiten representar el estado del sistema mediante la utilización de marcas (*tokens*, en inglés). Al estar conectadas las plazas con las transiciones, aquellas transiciones que tengan todas sus plazas de entrada con al menos una marca pasarán a estar activas y, por lo tanto, listas para ser ejecutadas. La ejecución de la transición eliminará las marcas de sus plazas de entrada y generará nuevas marcas en sus plazas de salida, cambiando así el estado de la red. La Figura 2.15 muestra gráficamente un ejemplo de marcado, activación y disparo de una transición.

Por otra parte, una red de Petri es un modelo abstracto y por ello puede representar diferentes situaciones. La Tabla 2.1 muestra algunas posibles interpretaciones de los elementos que forman una red. Por ejemplo, las plazas de entrada y salida de una transición pueden usarse para describir pre- y postcondiciones de un evento, y la transición la ocurrencia de dicho evento. Otra posible interpretación es que las plazas representen recursos y las transiciones actividades que utilizan y liberan recursos. En cualquiera de las posibles interpretaciones, la red de Petri se comportará siempre igual, independientemente de si las plazas representan datos, condiciones o antecedentes de reglas.

2.2.1.1. Variantes de redes de Petri

Las redes de Petri clásicas son extremadamente útiles para la especificación y simulación de procesos complejos que requieran paralelismo y concurrencia de procesos. Sin embargo, su complejidad crece significativamente cuando se utilizan para modelar procesos complejos. Por ello, se han creado distintas variantes que permiten modelar de forma más sencilla a sistemas más complejos. Las extensiones, comúnmente denominadas redes de Petri de alto nivel, más aplicadas al modelado de WFs son:

- *Redes de Petri coloreadas* [149]. Estas redes permiten modelar la identidad de las marcas asociándoles un valor denominado *color*. Esto facilita una descripción más detallada de los objetos utilizados en los WFs. La red de la Figura 2.16 muestra un ejemplo de red de Petri coloreada. En ella puede verse que las plazas están *tipadas*, de forma que únicamente puedan almacenar objetos de su tipo. Por ejemplo, las marcas a y b almacenadas en la plaza p_1 tienen el tipo $ColorA$ y se pueden utilizar para expresar condiciones, recursos o un simplemente un caso de un WF. Además, las transiciones y los arcos de la red están anotados con una expresión algebraica para razonar con esos colores:
  - Los arcos están anotados por multiconjuntos de términos, de modo que durante la evaluación de la expresión, cada uno de los elementos de ese multiconjunto deberá parearse con una de las marcas almacenadas en la red. Por ejemplo, el arco entre la plaza p_1 y la transición t_1 está anotado con el multiconjunto $x+y$. En este caso, tanto x como y son variables y podrán tomar durante la ejecución el valor $x = a$ e $y = b$ o $x = b$ e $y = a$. Los arcos también pueden estar anotados por términos más complejos como, por ejemplo, la función $f(x)$ entre la transición t_2 y la plaza p_3.
  - Las transiciones pueden tener condiciones de activación que se evalúan a partir de las marcas almacenadas en sus plazas de entrada. Por ejemplo, la transición t_1 está anotada con la precondición $(x = a \wedge x = b) \vee (y = a \wedge y = b)$.

- *Redes de Petri con tiempo* [288]. Las redes de Petri básicas no son lo suficientemente expresivas para estudiar el rendimiento de un sistema, ya que no realizan ninguna asunción acerca de la duración de las actividades del sistema. Respondiendo a esta necesidad, varias propuestas extendieron las redes de Petri con tiempo. Las redes de Petri con tiempo (TPN del inglés *Timed Petri Nets*). Estas redes pueden dividirse en dos clases: TPN *determinísticas*, en las cuales a cada transición, plaza o arco se le asocia un tiempo de ocurrencia o intervalo de tiempo determinístico, y las TPN *estocásticas*, en las cuales cada transición tiene asociado un tiempo de ocurrencia aleatorio. Tanto las TPN determinísticas como las estocásticas son aplicadas en un amplio rango de aplicaciones que, principalmente, apuntan a la evaluación de sistemas dinámicos de eventos discretos. Por ejemplo, las TPN determinísticas se han utilizado con éxito para derivar el tiempo de producción, identificar cuellos de botella, verificar

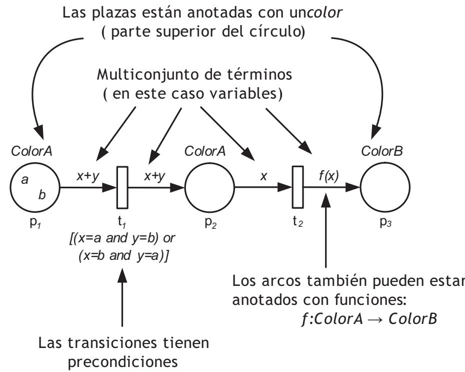


Figura 2.16: Ejemplo de red de Petri coloreada

restricciones temporales, etc; mientras que las TPN estocásticas también se han conseguido aplicar con éxito a la obtención de ratios de productividad, al análisis del rendimiento y de la demora, etc.

- *Redes de Petri jerárquicas* [149, 120]. Estas redes permiten componer redes más complejas a partir de otras más sencillas. Los elementos de composición suelen ser las plazas o las transiciones que ahora adoptan una semántica más compleja. Estos nodos pueden representar a otra red haciendo más fácil crear, mantener y entender modelos complejos. En el Capítulo 3 se detallarán los mecanismos que permiten crear y componer redes jerárquicas.

2.2.1.2. Ventajas de usar redes de Petri

Muchos investigadores han defendido el uso de las redes de Petri para el modelado de WFs [255, 98, 227]. Una de las principales ventajas de las redes de Petri respecto a otros formalismos es su legibilidad: al disponer de una representación gráfica relativamente sencilla, los diseñadores pueden crear sus procesos sin necesidad de apoyo por parte de expertos en redes de Petri. Además, la expresividad es otro de sus puntos fuertes. Son sistemas Turing completo (si se asume la no existencia de límites tecnológicos), lo cual implica que se pueden utilizar para emular máquinas de Turing y, por lo tanto, para realizar cualquier tipo de cálculo. Otras importantes razones son [255]:

1. *Al tener una semántica formal (i)* la especificación del WF no es ambigua, *(ii)* la interpretación está definida matemáticamente y, por lo tanto, no depende de la implementación de una herramienta, y *(iii)* es posible razonar acerca de las propiedades del WF especificado. Existen muchas técnicas de análisis para verificar ciertas propiedades cualitativas y cuantitativas de los modelos de WFs basados en redes de Petri. Dentro de las cualitativas destacan la comprobación

de la existencia de ciclos mortales, el análisis de la alcanzabilidad de cada transición, o la comprobación de los límites de marcas en cada una de las plazas. Respecto a las propiedades cuantitativas, es posible verificar algunas medidas de rendimiento como los tiempos de respuesta, de espera o los ratios de ocupación de los recursos.

2. Estas redes *modelan explícitamente el estado*, lo cual es de gran utilidad para el modelado de los WFs, ya que:

    - Permite diferenciar entre la activación y la ejecución de una tarea; una tarea debe estar activa antes de ejecutarse, sin embargo la activación no implica que vaya a ejecutarse necesariamente. Esta diferenciación posibilita que la ejecución de una tarea pueda esperar por un determinado evento, por ejemplo, una acción del usuario, un mensaje externo o simplemente un evento temporal.
    - Permite modelar la competición entre las tareas. Dos tareas que utilizan los mismos recursos (modelados como plazas) pueden estar activas al mismo tiempo; sin embargo, solamente una de ellas se ejecutará.
    - Permiten cancelar la ejecución de casos eliminando las marcas de las plazas.
    - En entornos distribuidos permite transferir los casos en ejecución de un sistema a otro simplemente transfiriendo las marcas de las plazas.
    - Permite que los sistemas sean reactivos a cambios en el entorno. Por ejemplo, definiendo eventos externos que cambien el estado de la ejecución.

    El modelado del estado es una gran diferencia con respecto a los marcos conceptuales basados en eventos (como π-calculus), los cuales modelan únicamente las transiciones entre los distintos estados, por lo que no pueden acceder al estado y por ello carecen de las ventajas descritas anteriormente.

2.2.1.3. Inconvenientes de usar redes de Petri

Aunque la expresividad es un punto fuerte de estas redes, esta afirmación no es relevante desde un punto de vista práctico, ya que se podría afirmar lo mismo de un lenguaje de programación: la clave no está sólo en la expresividad, sino también en lo complejo que sea lograrla. Por ello, aunque estas redes pueden representar cualquiera de los patrones de comportamiento descritos en el apartado 2.1.1.1, existen algunos inconvenientes a la hora de definir WFs mediante redes de Petri [261]:

- Aunque se puede usar el color para identificar múltiples instancias de un (sub)-proceso, por ejemplo asociando un identificador de la instancia de ejecución al color, no existe un soporte específico para patrones de instanciación múltiple. En estos casos, el diseñador debe soportar las redes para soportar el funcionamiento de los patrones de control a la instanciación múltiple.

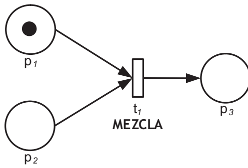


Figura 2.17: El patrón *mezcla simple* no puede resolverse con una transición normal

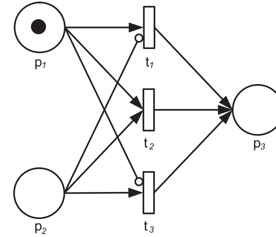


Figura 2.18: Solución al patrón *mezcla simple* utilizando arcos inhibidores

- Algunos patrones de sincronización son difíciles de modelar a través de redes de Petri debido a la regla de transición. Por ejemplo, el patrón de *mezcla simple* tiene un comportamiento contrario a la regla de transición. Para dar soporte a este patrón no vale una simple red como la representada en la Figura 2.17, ya que la regla de transición requiere que las dos plazas p_1 y p_2 tengan al menos una marca. Es, por lo tanto, necesario adecuar la red para dar soporte al comportamiento especificado en el patrón. Una solución es añadir *arcos inhibidores* a la red, en los que la ausencia de marcas es la que activa la transición, y no la presencia de marcas, como sucede en los arcos normales. Por ejemplo, en la Figura 2.18 puede apreciarse la representación del patrón *mezcla simple* utilizando arcos inhibidores (en lugar de una flecha tienen un círculo).

- La ocurrencia de una transición es siempre local: se basa en las marcas localizadas en las plazas de entrada y afecta a las plazas de salida. Sin embargo, en un WF algunos eventos pueden tener efectos que no son locales. Por ejemplo, un error puede implicar el eliminar marcas de un conjunto de plazas sin saber a priori en qué plazas están las marcas. El modelado de este tipo de patrón de cancelación es un proceso engorroso, ya que requiere conectar todas esas plazas al mecanismo de cancelación.

2.2.2. Pi-calculus

Pi-calculos (π-calculus) es un álgebra de procesos desarrollada por Robin Milner, Joachim Parrow and David Walker [180] que da continuación al cálculo de procesos CCS (del inglés *Calculus of Communicating Systems*) [179]. El objetivo de π-calculus es permitir la descripción de computaciones concurrentes cuya configuración puede cambiar durante la ejecución. Es un formalismo que permite la representación, simulación, análisis y verificación de sistemas móviles de comunicación. Un sistema representado mediante π-calculus, consta de múltiples procesos concurrentes que están agrupados

en parejas y que pueden enviar y recibir mensajes de forma sincronizada. Esta comunicación se realiza sobre canales de entrada y salida, de modo que cuando un proceso recibe un mensaje, también recibe el canal para enviar sus respuestas. Esta característica, llamada *movilidad*, permite que las conexiones de la red cambien con las interacciones realizadas, es decir, se producen cambios dinámicos en la topología de la comunicación.

Un proceso π-calculus es una entidad autónoma que posee puertos, para comunicarse con otros procesos, y que se representa de la siguiente forma:

$$proceso(lista\, de\, puertos) = comportamiento$$

De forma más detallada π-calculus define los siguientes elementos:

- Los nombres representan conceptos como los enlaces, los punteros, las referencias, los identificadores, etc. y donde cada nombre tiene un ámbito de aplicación.
- Los procesos se definen como:

  $$P \ ::= \ M \mid P \mid P' \mid (\mathbf{v}z)P \mid !P$$

  donde con M se indica la evaluación de una expresión; con $P \mid P'$ que los procesos P y P' son hebras ejecutadas de forma concurrente; con $\mathbf{v}zP$ que se aplicará una restricción $\mathbf{v}z$ al proceso P; y finalmente con $!P$ se representa que el proceso P será replicado, es decir, se ejecutará repetidas veces.
- Las expresiones pueden tomar los siguientes valores:

  $$M \ ::= 0 \mid \pi.P \mid M + M'$$

  donde el símbolo 0 representa un proceso especial, denominado *nil*, cuya ejecución es completa y ha finalizado; con $\pi.P$ se indica que la aplicación de un determinado prefijo al proceso P; y con $M + M'$ se representa una operación de elección entre los procesos M y M', es decir, que únicamente uno de los dos procesos se ejecutará.
- Los prefijos se indican como:

  $$\pi \ ::= \ \overline{x}y \mid x(z) \mid \tau \mid [x = y]\pi$$

  donde con $\overline{x}y$ se describe que el mensaje y se emitirá en el canal x (es decir, las entradas del proceso); con $x(z)$ se representa que el mensaje z se recibe en el canal de comunicaciones x (las salidas del proceso); con τ se indica una acción a realizar; y finalmente con $[x = y]\pi$ se describe la condición $[x = y]$ que el prefijo π debe cumplir.

La Figura 2.19 muestra la representación de un bucle tipo *ciclos arbitrarios* mediante una representación gráfica y su correspondiente representación en π-calculus. Puede apreciarse cómo cada uno de los rectángulos se corresponde con un proceso.

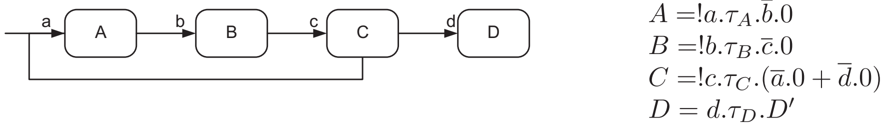


Figura 2.19: Representación de un bucle de tipo *ciclos arbitrarios* en π-calculus

2.2.2.1. Ventajas de usar π-calculus

Aunque en menor número que en el caso de las redes de Petri, muchos autores han defendido el uso de π-calculus para el modelado de WFs [234]. Con π-calculus también se pueden emular máquinas de Turing y, por lo tanto, implementar cualquier tipo de cómputo. Además, al ser un formalismo, tienen la ventaja de tener una semántica no ambigua. También disponen de herramientas de análisis, aunque en menor medida que las redes de Petri.

2.2.2.2. Inconvenientes de usar π-calculus

El principal inconveniente de π-calculus es su falta de legibilidad. π-calculus es un lenguaje basado en texto que no tiene una notación gráfica estándar y incluye un conjunto mínimo de constructores/primitivas. Ello fuerza al diseñador a construir largas y complejas cadenas de texto que pueden llevarle a la equivocación. Por ejemplo, la Figura 2.19 muestra un típico bucle *repetir-hasta* y su código π-calculus asociado. Como se puede apreciar en la imagen, la legibilidad de la representación basada en texto es mucho menor. Aunque se pueden definir construcciones más complejas partiendo de π-calculus [181], todo el esfuerzo recae en el diseñador que debe tener sólidos conocimientos de π-calculus. Algunos lenguajes extienden π-calculus y definen sus propias construcciones para facilitar este paso, pero el manejo de estos lenguajes sigue siendo exclusivo para programadores. Debido a este inconveniente, la mayoría de los lenguajes o sistemas de WFs que fundamentan su ejecución en este formalismo utilizan un lenguaje gráfico para facilitar la representación del WF, y establecen las correspondencias con π-calculus en [306]. Sin embargo, con esta combinación, aunque se gana en legibilidad, se pierde en expresividad, ya que π-calculus queda restringido a la capacidad expresiva del lenguaje gráfico.

Otro importante inconveniente de π-calculus es que no permite modelar el estado del WF. En este sentido, l os sistemas basados en π-calculus son reactivos, es decir, reaccionan ante eventos externos y no guardan el estado. Así, su expresividad es menor comparada con las redes de Petri, ya que no permite modelar los patrones de estado.

2.2.3. Máquinas de estados abstractos

Las máquinas de estados abstractos (ASMs, del inglés Abstract State Machines) son máquinas de estado desarrolladas por Gurevich [124] que permiten operar en estados

que manejan un conjunto de datos junto con una serie de funciones y relaciones que operan sobre los datos. Las ASMs tienen una base matemática sencilla y proporcionan una forma intuitiva de *pseudo*-código sobre datos abstractos. Por ello, la principal aplicación de este formalismo consiste en el modelado e implementación de la semántica de lenguajes como Prolog [39], C++ [287], Java [40] o BPEL [102]. Sin embargo, en los últimos años este método se ha aplicado a varios proyectos relacionados con el modelado de servicios web, WFs y procesos de negocio [41].

Una máquina secuencial ASM se define como un conjunto de reglas de transición con la siguiente forma:

if Condition **then** Updates

donde:

- Condition es una fórmula de primer orden, libre de variables, que representa la condición de guardia que debe satisfacerse para que la regla sea aplicable.
- Update es un conjunto finito de funciones $f(t_1, \ldots, t_n) := t$ donde los términos t_i no contienen variables.

En cada estado, todas las reglas aplicables se ejecutan simultáneamente para producir el siguiente estado. Además, también introduce instrucciones típicas de lenguajes funcionales como **let** ... **in**, para dar soporte al no determinismo:

choose x**with** ϕ rule

o para tratar el paralelismo síncrono en la ejecución de reglas ASMs:

forall x **with** ϕ **do** rule

forall x **with** ϕ **until** rule

El siguiente código muestra la representación de un bucle tipo *ciclos arbitrarios* mediante ASMs:

forall $a \in$ Activity Iterate(a) until StopCriterion(a)

La notación que aparece está simplificada de acuerdo a la sintaxis introducida en [37].

2.2.3.1. Ventajas de usar ASMs

En los últimos años, varios autores han defendido el uso de las ASMs para el modelado de WFs [37]. Este formalismo ha probado su validez para el modelado y validación de la semántica operacional de varios lenguajes de programación y en lenguajes de procesos como por ejemplo BPEL [102]. Otra aportación relevante es la semántica operacional en ASMs para el modelado de procesos descritos mediante el lenguaje BPMN [41]. Como veremos en el apartado 2.3.1, BPMN (Business Process Modeling Notation) es el estándar OMG para la representación gráfica de procesos.

Las ASMs tienen la ventaja de ser muy expresivas, tener una semántica no ambigua y de disponer de herramientas para la validación. Respecto a este último punto, destacan en la detección de errores en las fases iniciales del desarrollo, donde la verificación, simulación y la prueba son menos costosas de aplicar.

2.2.3.2. Inconvenientes de usar ASMs

Al igual que π-calculus, el principal problema de las ASMs es su falta de legibilidad. Además, su legibilidad se reduce a medida que se añaden reglas al modelo. Debido a este problema, este formalismo únicamente se usa para dar soporte a la ejecución de WFs. Los sistemas que lo implementan usan por ello un lenguaje de representación gráfica para facilitar la creación de los WFs y establecen las correspondencias con la semántica de las ASMs.

2.3. Lenguajes de representación

Aunque no existan lenguajes formales creados específicamente para modelar WFs, sí existen modelos de representación directamente pensados para WFs. Algunos son modelos gráficos pensados exclusivamente para facilitar la labor del diseño, otros son adaptaciones de modelos tradicionalmente utilizados en la ingeniería del software, y finalmente unos pocos están ideados dentro de un escenario de intercambio electrónico de WFs. Algunos de estos lenguajes están por completo orientados a la descripción gráfica y, por lo tanto, carecen de una interpretación que les permita ser ejecutables. Sin embargo, a medida que estos modelos han adquirido importancia, han aparecido traductores que permiten trasladar la representación gráfica de estos lenguajes a un modelo u otro lenguaje ejecutable. Dentro de esta categoría las principales aportaciones son los lenguajes BPMN, UML y XPDL.

2.3.1. BPMN

El lenguaje BPMN [197] (del inglés *Bussiness Process Modeling Notation*) es un estándar de notación de proceso publicado en el año 2003 por el Object Management Group[2]. Permite definir diagramas de procesos BPD (del inglés *Business Process Diagram*), que tienen como objetivo crear un modelo gráfico de las operaciones del proceso de negocio. Esto ha propiciado que BPMN haya sido combinado con otros lenguajes que pueden interpretar su representación gráfica como BPML, XPDL o BPEL [197, 232]. La Figura 2.20 muestra las categorías y los elementos básicos de BPMN:

- *Objetos de flujo*. Un BPD tiene únicamente tres componentes con los que se puede modelar un flujo de trabajo:

[2]http://www.bpmi.org

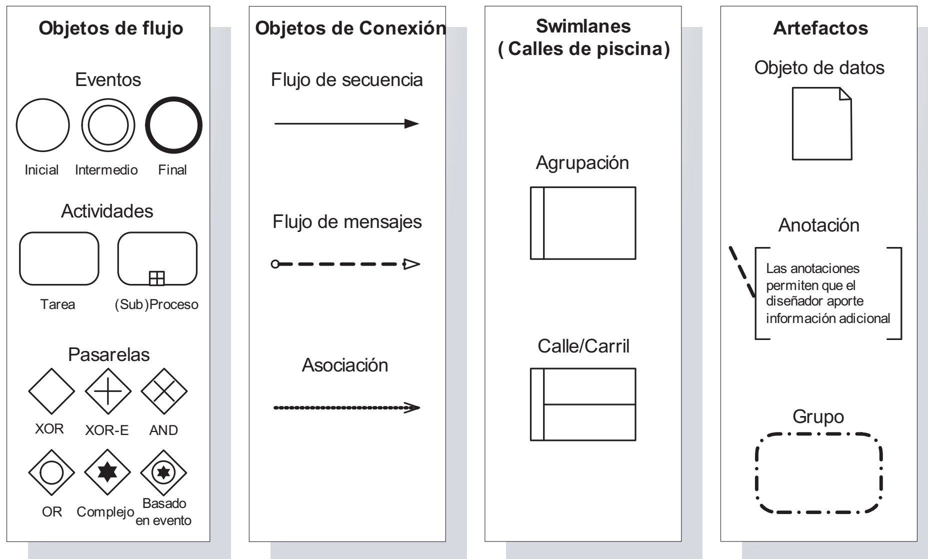


Figura 2.20: Elementos básicos del lenguaje BPMN

- *Eventos*. Representan un suceso en el curso del proceso de negocio. Afectan al flujo del proceso y normalmente tienen una causa (*disparo*) y un impacto (*resultado*). Existen tres tipos de eventos en función del momento en el que afecten al flujo: iniciales, intermedios y finales.
- *Actividades*. Es un término genérico para referirse al trabajo a realizar. Cuando una actividad es atómica se denomina *tarea* mientras que si es compuesta se denomina (sub)proceso.
- *Pasarelas*. Son los componentes o construcciones de control. BPMN contiene seis componentes: *xor*, *xor explícito*, *and*, *or*, *complejo* y *basado en evento*.

- *Objetos de conexión*. Los objetos de flujo están interconectados para definir la estructura del proceso de negocio. BPMN define para ello tres tipos de objetos de conexión:

  - *Secuencia de flujo*. Se utiliza para mostrar el orden (*secuencia*) de actividades que se van a realizar en el proceso.
  - *Flujo de mensajes*. Muestra el envío y recepción de mensajes entre dos participantes del proceso (entidades o roles de negocio). Dichos participantes se representarán por medio de dos agrupaciones diferentes.
  - *Asociación*. Se utiliza para asociar datos, texto u otro artefacto con los objetos de flujo, permitiendo mostrar las entradas y salidas de las actividades.

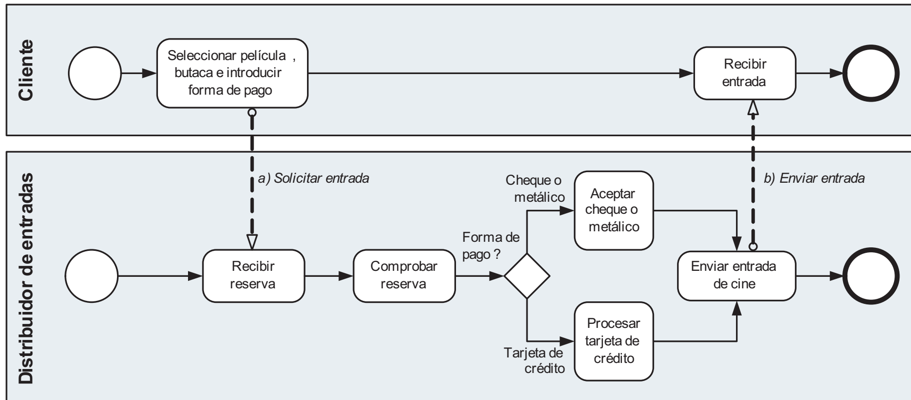


Figura 2.21: Ejemplo simplificado de compra de una entrada de cine representada mediante un diagrama BPMN

- *Calles o carriles de piscina* (*swimlanes*, en inglés). Muchas metodologías de modelado de procesos utilizan el concepto de *swimlane* para organizar las actividades en categorías separadas, de forma que puedan ilustrar sus diferentes funciones o responsabilidades. BPMN soporta este concepto mediante dos construcciones:
  - *Agrupación* (*pool*, en inglés). Representa un participante del proceso, aunque también puede actuar como un contenedor de actividades.
  - *Calle o carril* (*lane*, en inglés). Es una (sub)partición dentro de una agrupación, y se utilizan para organizar y categorizar actividades.
- *Artefactos*. Son elementos cuya funcionalidad es dar flexibilidad de notación para identificar o describir elementos gráficos. BPMN define tres componentes principales, aunque esta parte del estándar está abierta:
  - *Objeto de datos*. Son un mecanismo para mostrar los datos requeridos o producidos por una actividad. Se conectan a las actividades mediante asociaciones.
  - *Anotación*. Permite al diseñador añadir información adicional acerca del diagrama.
  - *Grupo*. Se utiliza para agrupar elementos de cara a mejorar la documentación o el análisis, pero no tiene un significado dentro del proceso ni afecta al flujo de secuencia.

La Figura 2.21 muestra un ejemplo de reserva de entradas de cine representado mediante BPMN. El diagrama contiene dos agrupaciones para distinguir los dos roles del modelo, es decir, los clientes y el distribuidor de entradas. Dentro de la agrupación del cliente puede verse cómo selecciona la película y la butaca, e introduce la forma

Tabla 2.2: Expresividad de los lenguajes de representación y ejecución de WFs (I) [260]: 1-BPMN, 2-UML, 3-XPDL, 4-BPEL, 5-BPML, 6-YAWL, 7-OWL-S, 8-WSMO. Los signos '+' indican un soporte directo, los '+/-' un soporte parcial y los '-' que el lenguaje no da soporte al patrón. Un '-' no implica que sea imposible representar el patrón, sino que la solución no es directa. De hecho todos estos patrones se podrían resolver a través de programación pero éste no es el objetivo de los lenguajes de WFs.

Nombre del patrón	1	2	3	4	5	6	7	8
Control básico								
Secuencia	+	+	+	+	+	+	+	-
Separación	+	+	+	+	+	+	+	-
Sincronización	+	+	+	+	+	+	+	-
Elección exclusiva	+	+	+	+	+	+	+	-
Mezcla	+	+	+	+	+	+	-	-
Control avanzado								
Elección múltiple	+	+	+	+	-	+	-	-
Mezcla sincronizada estr.	+	-	+	+	-	+	-	-
Mezcla sincronizada local	-	+/-	-	+	-	+	-	-
Mezcla sincronizada general	-	-	-	-	-	+	-	-
Mezcla múltiple	+	+	+	-	+/-	+	-	-
Discriminador estructurado	+/-	+/-	+/-	-	-	+	-	-
Bloqueando discriminador	+/-	+/-	+/-	-	-	+	-	-
Cancelando discriminador	+	+	+	-	-	+	-	-
Unión parcial estructurada	+/-	+/-	+/-	-	-	-	-	-
Bloqueo unión parcial	+/-	+/-	+/-	-	-	-	-	-
Cancelación unión parcial	+/-	+	+/-	-	-	-	-	-
Unión generalizada	+	-	+	-	-	-	-	-
Mezcla de hilos de ejecución	+	+	+	+/-	-	-	-	-
Separación de hilos de ejecución	+	+	+	+/-	-	-	-	-
Instanciación múltiple (IM)								
Sin sincronización	+	+	+	+	+	+	-	-
Definida en tiempo de diseño	+	+	+	-	-	+	-	-
Definida en tiempo de ejecución	+	+	+	-	-	+	-	-
Sin conocimiento previo	-	-	-	-	-	+/-	-	-
Unión parcial estática IM	+/-	-	+/-	-	-	-	-	-
Unión parcial dinámica IM	-	-	-	-	-	-	-	-
Cancelación unión parcial IM	+/-	-	+/-	-	-	-	-	-

de pago antes de comunicarse con el distribuidor. Una vez que el distribuidor recibe la solicitud del cliente, comprueba la reserva y aplica la forma de pago. Finalmente, el distribuidor le pasa las entradas al cliente.

2.3.1.1. Ventajas de usar BPMN

Dentro de los elementos más positivos de BPMN destaca su legibilidad, ya que intenta definir una notación común inteligible tanto por los diseñadores como por los expertos [301]. Gracias a este esfuerzo BPMN ha ganado terreno en relación a UML como el lenguaje de notación gráfica de WF por excelencia.

Las tablas 2.2 y 2.3 representan la expresividad de este lenguaje respecto a los patrones de comportamiento descritos en el apartado 2.1.1.1. Se puede observar cómo

Tabla 2.3: Expresividad de los lenguajes de representación y ejecución de WFs (II) [260]: 1-BPMN, 2-UML, 3-XPDL, 4-BPEL, 5-BPML, 6-YAWL, 7-OWL-S, 8-WSMO.

Nombre del patrón	1	2	3	4	5	6	7	8
Estado								
Elección diferida	+	+	+	+	+	+	-	-
Enrutamiento paralelo alternado	-	-	-	+/-	-	+	-	-
Hito	-	-	-	-	-	+	-	-
Sección crítica	-	-	-	+	-	+	-	-
Enrutamiento alternado	+/-	-	+/-	+	+	+	-	-
Cancelación								
Cancelar actividad	+	+	+	+	+	+	-	-
Cancelar caso	+	+	+	+	+	+	-	-
Cancelar región	+/-	+	+/-	+/-	+/-	+	-	-
Cancelar actividad de MI	+	+	+	-	-	+	-	-
Completar actividad de MI	-	-	-	-	-	-	-	-
Iterativos								
Ciclos arbitrarios	+	+	+	-	-	+	-	-
Bucles estructurados	+	+	+	+	+	-	+	-
Recursión	-	-	-	-	-	-	+	-
Terminación								
Terminación implícita	+	+	+	+	+	+	+	-
Terminación explícita	+	+	+	-	-	+	-	-
Basado en eventos								
Disparo transitorio	-	+	-	-	-	-	-	-
Disparo permanente	+	+	+	+	+	-	-	-

el lenguaje cumple la mayor parte de estos patrones, aunque tiene problemas para expresar algunos patrones de control avanzado, de estado y de instanciación múltiple [302].

Respecto a los patrones de datos, tal y como se muestra en la Tabla 2.4, BPMN es el lenguaje con mejor valoración y soporta 18 de los 32 patrones. Así, permite asociar a los datos una visibilidad a nivel de tarea, de casos de ejecución y de bloque al soportar la descomposición de las tareas. Este hecho facilita que además de la interacción entre tarea-tarea y tarea-entorno, también permita una interacción entre tarea-(sub)tarea. En cuanto a la transferencia de datos, soporta el pase de parámetros tanto por valor como por referencia, e incluso facilita la transformación de los datos antes de llegar o después de ejecutarse la tarea. Finalmente, soporta todos los patrones de enrutamiento con la excepción de la verificación de las pre- y postcondiciones basadas en el valor de los datos.

Por último, BPMN también permite representar algunos de los patrones de organización identificados en el apartado 2.1.1.3. La Tabla 2.5 muestra cómo BPMN y UPML son los únicos lenguajes que dan un soporte real a estos patrones. Si bien este soporte es reducido, 9 de los 43 patrones, proporciona la posibilidad de realizar una asignación directa de recursos o basada en roles.

Tabla 2.4: Patrones de datos soportados por los lenguajes de representación y ejecución de WFs [218]. Los lenguajes BPML, YAWL y WSMO no se incluyen en la comparación, ya que no dan un soporte directo a este tipo de patrones.

Nombre del patrón	BPMN	UML	XPDL	BPEL	OWL-S
Visibilidad					
Tarea	+	+/-	-	+/-	-
Bloque	+	+	+	-	-
Ambito	-	-	-	+	-
Múltiples instancias	+/-	+	+	-	-
Caso	+	-	+	+	+
Carpeta	-	-	-	-	-
Flujos de trabajo	-	+	+/-	-	-
Entorno	-	-	-	+	-
Interacción					
Tarea a tarea	+	+	+	+	+
Bloque a (sub)flujo de trabajo	+	+	+	-	+
(Sub)flujo de trabajo a bloque	+	+	+	-	+
A instancias múltiples	-	+	-	-	-
De instancias múltiples	-	+	-	-	-
Caso a caso	-	-	+/-	+/-	-
Tarea a entorno (subir)	+	-	+	+	+
Tarea a entorno (bajar)	+	-	-	+/-	-
Entorno a tarea (subir)	+	-	-	+/-	-
Entorno a tarea (bajar)	+	-	+	+	-
Caso a entorno (subir)	-	-	-	-	-
Caso a entorno (bajar)	-	-	-	-	-
Entorno a caso (subir)	-	-	-	-	-
Entorno a caso (bajar)	-	-	-	-	-
WF a entorno (subir)	-	-	-	-	-
WF a entorno (bajar)	-	-	-	-	-
Entorno a WF (subir)	-	-	-	-	-
Entorno a WF (bajar)	-	-	-	-	-
Transferencia					
Por valor (entrante)	+	-	+/-	+	-
Por valor (saliente)	+	-	+/-	+	-
Copia entrada/Copia Salida	+/-	-	+/-	-	-
Por referencia sin bloqueo	-	-	+	+	+
Por referencia con bloqueo	+	+	-	+/-	-
Transformación de entrada	+/-	+	-	-	-
Transformación de salida	+/-	+	-	-	-
Enrutamiento					
Precondición - existencia dato	+	+	-	+/-	+
Precondición - valor dato	-	+	+	+	+
Postcondición - existencia dato	+	+	-	-	+
Postcondición - valor dato	-	+	-	-	+
Disparo de tarea por evento	+	+	-	+	-
Disparo de tarea por dato	+	-	-	+/-	-
Enrutamiento basado en datos	+	+	+	+	+/-

2.3.1.2. Inconvenientes de usar BPMN

A pesar de ser un lenguaje estándar, BPMN no tiene un formato de fichero estandarizado para almacenar modelos BPMN, por lo que el formato puede variar de unos

Tabla 2.5: Patrones de organización soportados por los lenguajes de representación y ejecución de WFs [224]: 1-BPMN, 2-UML, y 3-XPDL.

Nombre del patrón	1	2	3	Nombre del patrón	1	2	3
Creación				Asignación			
Directa	+	+	+	Inicada por recursos (IR)	-	-	-
Basada en roles	+	+	+	IR - trabajos asignados	-	-	-
Diferida	-	-	-	IR - trabajos ofrecidos	-	-	-
Autorización	-	-	-	Cola determinada por sistema	-	-	-
Separación responsabilidades	-	-	-	Cola determinada por recursos	-	-	-
Manejo de casos	-	-	-	Selección autónoma	-	-	-
Casos familiares	-	-	-	Rodeo			
Habilidad	-	-	-	Delegación	-	-	-
Historial	-	-	-	Intensificación	-	-	-
Organización	-	-	-	Desasignación	-	-	-
Ejecución automática	+	+	+	Redistribución	-	-	-
Ofrecimiento				Stateless Reallocation	-	-	-
Ofr. a único recurso	-	-	-	Suspensión/Resumen	-	-	-
Ofr. a múltiples recursos	-	-	-	Salto	-	-	-
Asignación a único recurso	+	+	+	Rehacer	-	-	-
Asignación aleatoria	-	-	-	Pre-hacer	-	-	-
Asignación round robin	-	-	-	Comenzar al crear	+	+	+
Cola más corta	-	-	-	Comenzar al asignar	-	-	-
Distribución temprana	+	+	+	Ejecución apilada	-	-	-
Distribución en activación	+	+	+	Ejecución encadenada	+	+	+
Distribución tarde	-	-	-	Visibilidad trabajo sin asignar	-	-	-
				Visibilidad trabajo asignado	-	-	-
				Ejecución simultánea	+	+	+
				Recursos adicionales	-	-	-

sistemas a otros. En cualquier caso, el principal inconveniente de BPMN es su ambigüedad, ya que no está formalizado y, por lo tanto, llevar al diseñador a asumir una semántica incorrecta. Incluso las correspondencias entre BPMN y BPEL están definidas en texto y por ello sujetas a la interpretación del programador. Debido a ello, la semántica de ejecución de BPMN puede variar en función de las correspondencias definidas con los lenguajes de ejecución.

Una última observación que se puede deducir del análisis de las tablas 2.2, 2.3, 2.4 y 2.5 es la siguiente: los lenguajes de representación tienen una mayor expresividad que los lenguajes de ejecución. Aunque BPML, UML y XPDL suelen integrarse habitualmente con BPEL, sin embargo, ¿cómo podría BPEL interpretar modelos que incluyen patrones que no soporta?, como la *mezcla múltiple*, todos los tipos de *discriminadores*, las *uniones parciales* o los *ciclos arbitrarios*. Una posible respuesta es la que se indica en [198]: "los lenguajes de modelado tienden a tener mejores evaluaciones que los lenguajes operacionales por no tener que cargar con la implementación de todo aquello que son capaces de modelar".

2.3.2. Diagramas de actividad de UML

Dentro de la iniciativa UML (del inglés *Universal Modeling Language*) [10] se han creado sistemas de notación gráfica que han sido aplicados a la representación de WFs [91]. El Object Management Group ha adaptado los diagramas de estado dentro de la versión 2.0 del estándar UML para crear los llamados *diagramas de actividad* [215]. Estos diagramas se pueden utilizar para representar gráficamente secuencias de actividades y, al igual que BPMN, incorporan el concepto de *calle de piscina* para diferenciar los distintos roles que participan en la ejecución del WF. La Figura 2.22 muestra los elementos básicos de los diagramas de actividad de UML:

- *Nodo inicial.* Representa el punto de partida del flujo de actividades y es único dentro del diagrama. Se define dentro de la calle de piscina correspondiente al rol que comienza el caso de uso.
- *Nodo final.* Indica el final del flujo de actividades y, al contrario que en el caso del nodo inicial, en el diagrama se permite la existencia de más de un nodo de este tipo. Cada uno de estos nodos indicará una forma de concluir el caso de uso.
- *Actividad.* Tarea/Actividad genérica que realiza algún trabajo.
- *Transición.* Señala el orden de ejecución de las actividades.
- *Decisión.* Al igual que en los diagramas de flujo, las decisiones se denotan mediante un diamante, con las opciones marcadas en los arcos que salen del diamante.
- *Señal.* Es una actividad que manda o recibe un mensaje. Existen dos tipos de señales: de entrada (polígono cóncavo) y de salida (polígono convexo).
- *Actividad concurrente.* Algunas actividades pueden ocurrir simultáneamente o en paralelo. Estas actividades se denominan *actividades concurrentes*. Por ejemplo, escuchar al profesor y mirar el encerado es una actividad paralela.

La Figura 2.23 muestra el mismo WF de reserva de entradas de cine que fue introducido en el apartado de BPMN, pero en este caso modelado a través de un diagrama de actividades UPML. A simple vista la expresividad y legibilidad de ambos lenguajes es muy parecida.

Los diagramas de actividad de UML son un caso especial de diagrama de estado, que a su vez son una representación gráfica de las máquinas de estado. El formalismo de máquinas de estado en el que está parcialmente basado UML es una variante de los diagramas de Harel [127] y representan sistemas de transiciones tipo *evento-condición-acción*. En estos sistemas la ocurrencia de un evento ejecuta una transición si *(i)* la máquina está en el estado inicial de dicha transición, *(ii)* el tipo de evento coincide con la descripción de la transición y *(iii)* la condición de la transición se verifica. Tanto la parte del evento (también llamado *disparador*), de la condición

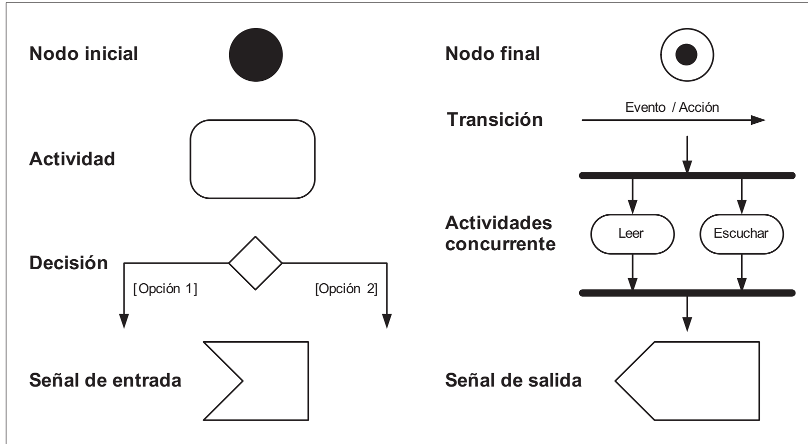


Figura 2.22: Elementos básicos de los diagramas de actividad del lenguaje UML

(también llamada *guardia*) como de la acción son opcionales en estos diagramas. Además, UML distingue los siguientes estados simples:

- *Estado de espera* (*wait-state*, en inglés). No se realiza ninguna acción o actividad.
- *Estado de acción* (*action-state*, en inglés). Una única acción está asociada al estado, y su ejecución es atómica.
- *Actividad en estado* (*activity-in-state*, en inglés). Una actividad, expresada como una secuencia de acciones, está asociada al estado. Aunque en teoría la ejecución de esta actividad puede abortarse, el estándar no especifica la forma de realizar dicho procedimiento.

Además, el estado puede contener la máquina de estados completa o estar compuesto por el estado de distintas actividades. A este respecto, UML proporciona dos tipos de estados compuestos: *AND-state* y *OR-state*. Para estados *AND-state*, concluir la ejecución requerirá completar el conjunto de estados que conforman la región, mientras que para estados *OR-state* sólo se requerirá la finalización de uno de ellos.

2.3.2.1. Ventajas de usar UML

Al igual que BPMN, la principal ventaja de UML es su legibilidad. En cuanto a la expresividad, los diagramas de actividad de UML soportan la mayoría de los patrones de comportamiento de WFs tal y como se muestra en las tablas 2.2 y 2.3. En este sentido, son tan expresivos como BPMN aunque este último permite representar parcialmente un par de patrones más [300]. Por esta razón, sigue siendo muy utilizado debido a la influencia de UML en el desarrollo del software, y es por ello muy

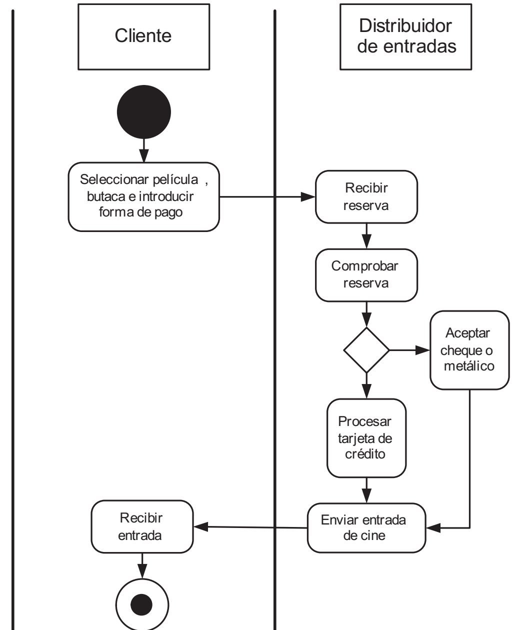


Figura 2.23: Ejemplo simplificado de compra de una entrada de cine representada mediante un diagrama UML

conocido tanto por desarrolladores como diseñadores. Con respecto a otros lenguajes de representación, los puntos fuertes de estos diagramas son [215]:

- Soporte del concepto de envío y recepción de señal al nivel conceptual.
- Soporte de la descomposición de una actividad en (sub)actividades (BPMN y XPDL también soportan esta característica).

La combinación de estas dos propiedades es un poderoso mecanismo para manejar la interrupción de actividades y con ello facilitar el desarrollo de los patrones de cancelación.

Con respecto a los patrones de datos, UML soporta datos con visibilidad de tarea, de bloque, instancias múltiples y de WF. El único problema de UML respecto a este tipo de patrones es que no da cuenta de la visibilidad de caso, sobre todo teniendo en cuenta que los WFs están diseñados para dar soporte a *casos* de ejecución. El hecho de que el principal propósito de los diagramas de actividad no sea el modelado de WFs puede ser una causa de este problema. Además, UML permite modelar la interacción entre tareas, (sub)tareas (bloques) e instancias múltiples. Desde el punto de vista de la transferencia, al estar basado en la teoría de objetos, sólo permite el pase de parámetros por referencia. Finalmente, respecto a los patrones de enrutamiento, UML es el más completo de los lenguajes y únicamente no soporta el disparo de tareas

en base a valores de datos (la dinámica de la orientación a objetos está guiada por eventos).

UML también da soporte a los mismos patrones de organización que BPML, es decir, permite una asignación directa y basada en roles (especificados a través de las calles de piscina).

2.3.2.2. Inconvenientes de usar UML

A pesar del marco formal proporcionado por los diagramas de Harel [127], las características específicas de los diagramas de actividad de UML sólo están parcialmente formalizadas (a través de expresiones OCL) y su descripción es en lenguaje natural, dejando ambiguos muchos conceptos [91]. Aunque la definición de una semántica precisa de UML es sujeto de un intenso debate e investigaciones, los diagramas de actividades han recibido poca atención a este respecto [100, 38]. Además, algunos inconvenientes de los diagramas de actividad son:

- Algunos constructores no tienen una sintaxis ni una semántica precisa.
- No permiten representar algunos patrones básicos de sincronización como los discriminadores o las uniones múltiples.
- No tienen un formato de fichero estándar, de forma que es difícil trasladar su semántica gráfica a un lenguaje de ejecución.

2.3.3. XPDL

El lenguaje XPDL [295] (del inglés, *XML Process Definition Language*) es un estándar de intercambio de ficheros definido por la Workflow Management Coalition[3] (WfMC). Este modelo fue inicialmente diseñado para intercambiar el diseño de procesos de negocio entre distintas herramientas. El punto de partida de XPDL es un conjunto mínimo de construcciones presente en la mayoría de productos de WFs. El metamodelo de procesos de XPDL está representado en la Figura 2.24 y está compuesto por los siguientes elementos:

- *Aplicación* (clase *Application*). Permite especificar aplicaciones/herramientas externas invocadas desde los procesos.
- *Proceso* (lase *Process*). Permite definir procesos o partes de un proceso, que se compone de dos tipos de elementos:
    - *Actividad* (clase *Activity*) es el componente principal de un proceso y se refiere a la realización de algún trabajo. Hay cinco tipos de actividades:

[3] http://www.wfmc.org

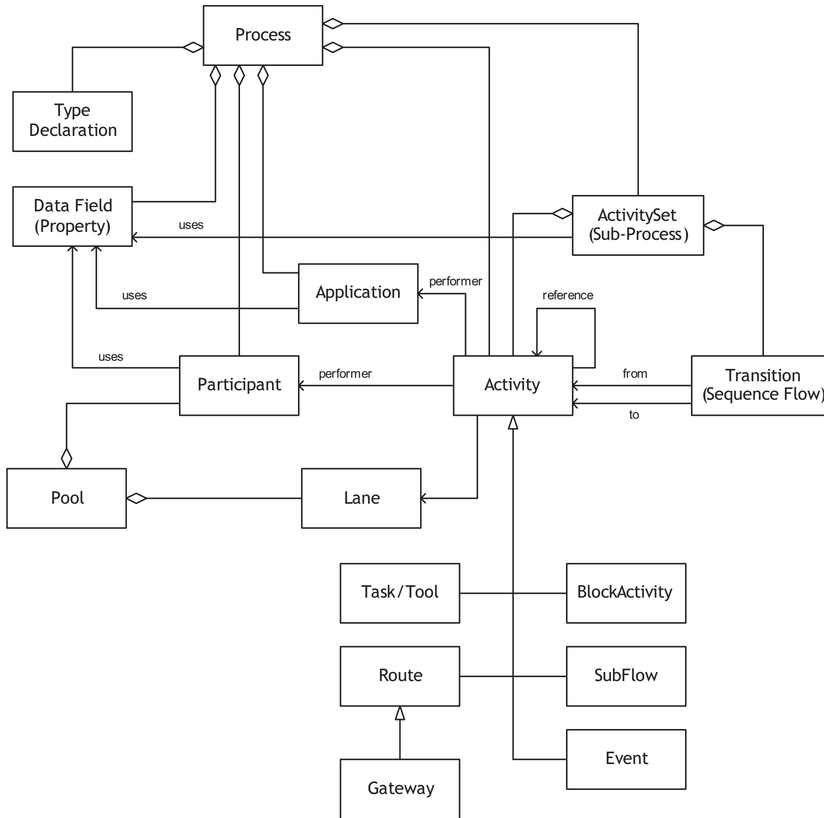


Figura 2.24: Metamodelo de proceso de XPDL

- ∘ Las *rutas* (clase *Route*) son actividades vacías que no realizan ningún trabajo y que simplemente se utilizan con propósitos de enrutamiento.
- ∘ Las *implementaciones* (clase *Task/Tool*) son actividades atómicas que se implementan con procedimientos automáticos o manuales y que no se descomponen.
- ∘ Los *bloques* (clase *BlockActivity*) son un conjunto o agrupación de actividades y/o transiciones. Todos los elementos del bloque comparten el mismo espacio de nombres.
- ∘ Los *(sub)flujos* (clase *SubFlow*) son contenedores para la ejecución de otro proceso. Este proceso puede ejecutarse localmente dentro del mismo servicio o (a través de la interfaz de interoperabilidad) en un servidor remoto y define sus propias actividades, transiciones, recursos y asignaciones.
- ∘ Los *eventos* (clase *Event*) afectan a la ocurrencia/disparo de la actividad y a la ruta que puede tomar.

- *Transición* (clase *Transition*) permite conectar actividades. Se usan para

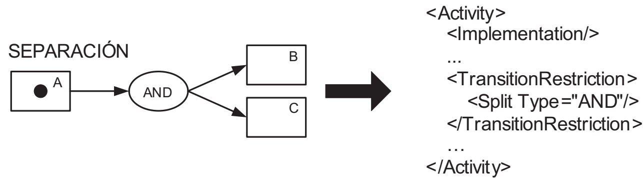


Figura 2.25: Representación en XPDL de una separación

establecer restricciones y condiciones como por ejemplo separar o sincronizar actividades o definir actividades concurrentes. Las restricciones (clase *TransitionRestriction*) son las que manejan el enrutamiento entre las actividades. Dentro de esta clase podemos destacar a las clases *Split* y *Join* que dan soporte a la separación y sincronización de actividades. Como se puede apreciar en la parte derecha de la Figura 2.25, el comportamiento de la separación (o de la unión) se establece a través del atributo *Type* que puede tomar los valores *AND*, *OR* y *XOR*.

- *Participante* (clase *Participant*). Especifica los participantes del WF, es decir, las entidades encargadas de realizar el trabajo. Existen seis tipos de participantes: *conjunto de recursos*, *recurso*, *rol*, *unidad organizativa*, *humano* y *sistema*. Al igual que en BPMN, los participantes del WF se introducen en el modelo dentro de calles de piscina (*lane*, en inglés).
- *Campo de datos* (clase *Datafield*) y *Declaraciones de tipo* (clase *TypeDeclaration*). Especifican los datos relevantes del WF. Estos datos se utilizarán para la toma de decisiones, para referirse a datos externos al WF y para el intercambio de información entre actividades y (sub)flujos.

2.3.3.1. Ventajas de usar XPDL

Es un lenguaje bastante utilizado y está presente en la mayoría de las herramientas de WFs. La razón es simplemente práctica: a pesar de tener menos expresividad que los demás lenguajes de representación, posee un conjunto suficiente de elementos para especificar la gran mayoría de los WFs. Además, el lenguaje está pensado para el intercambio de WFs, y por ello, no solo especifica la coordinación entre procesos, sino que también trata aspectos gráficos como el tamaño y las coordenadas de los elementos del modelo. En este sentido, XPDL facilita la integración de WFs entre herramientas a través de un formato de intercambio basado en XML.

2.3.3.2. Inconvenientes de usar XPDL

Deafortunadamente, con el conjunto mínimo de construcciones presente en XPDL no es posible representar un buen número de patrones de WFs que hoy en día están presentes en muchos de los productos de WFs. Como se aprecia en las tablas 2.2

y 2.2, XPDL tiene casi la misma expresividad que BPMN y que los diagramas de actividad de UML. Para mejorar este aspecto, productos que usan XPDL, como Enhydra Shark[4], Bonita[5] o WfMOpen[6], ofrecen extensiones no estándar y utilizan su propio formato de XPDL extendido. Sin embargo, esta aproximación no está alineada con la filosofía de XPDL que pretende ser una *lingua franca* para WFs.

Además del problema de expresividad, XPDL presenta un problema semántico. La WfMC no proporciona una especificación formal y sin ambigüedades de todos los elementos del lenguaje. Como resultado, muchos productos dicen ser compatibles con XPDL cuando en realidad interpretan algunos constructores de diferente manera [256]. Tampoco la legibilidad es un aspecto a destacar: los WFs están representados en XML y por ello pensados para ser interpretados por una herramienta. Por este motivo, XPDL suele combinarse con BPMN, aunque con menor expresividad y así tener una representación gráfica con la que se mejora su legibilidad.

Si se compara con BPMN o UML, soporta menos patrones de datos y los mismos patrones de organización.

2.4. Lenguajes de ejecución

Debido a las dificultades de definir un modelo de ejecución propio asociado a cada uno de los lenguajes de representación vistos en el apartado anterior, muchos investigadores han abordado el problema de la ejecución creando nuevos lenguajes. La mayoría de estos lenguajes se basan en alguno de los formalismos de representación previamente descritos en el apartado 2.2, para asegurar que su ejecución no sea ambigua. Dentro de esta categoría las principales aportaciones son los lenguajes BPML [247], BPEL [15] y YAWL [259].

2.4.1. BPML

BPML (*Business Process Modeling Language*) [247] es un estándar desarrollado y promovido por BPMI.org (*Business Process Management Initiative*). Los principales componentes de BPML, representados en la Figura 2.26, son:

- *Actividades* (clase *Activity*). Son componentes que realizan alguna funcionalidad. BPML define dos tipos de actividades:
  - *Actividades simples* (clases *AtomicActivity*. No se pueden descomponer y son atómicas. Una actividad simple puede ser de uno de los siguientes tipos:
    - *Acción*. Realiza una operación que implica el intercambio de mensajes de entrada y salida y la posterior actualización de propiedades.

[4] http://www.enhydra.org/index.php
[5] http://wiki.bonita.objectweb.org
[6] http://wfmopen.sourceforge.net

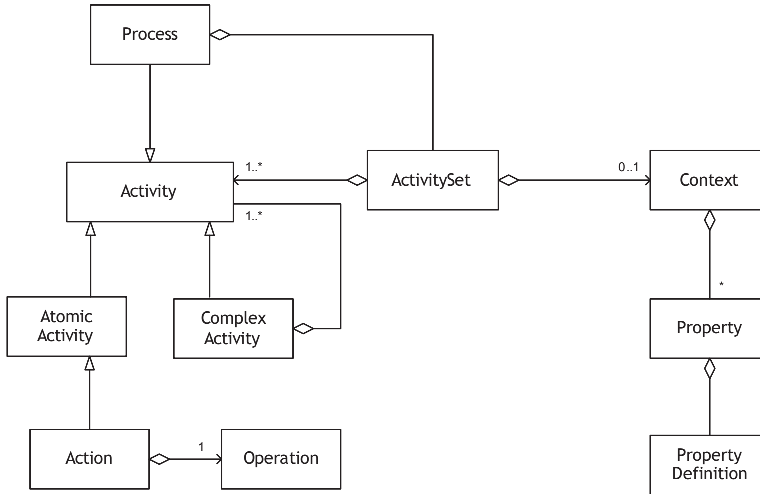


Figura 2.26: Metamodelo de proceso de BPML

- *Asignación*. Escribe un valor nuevo a una propiedad.
- *Llamada*. Instancia un proceso y espera a que se complete.
- *Compensación*. Invoca una acción de reparación (o compensación) para un determinado proceso.
- *Retraso*. Espera un determinado período de tiempo.
- *Vacío*. No realiza nada.
- *Fallo*. Lanza un fallo en el contexto actual.
- *Lanzar*. Lanza una señal.
- *Generar*. Instancia un proceso y no espera a que finalice.
- *Sincronizar*. Espera la ocurrencia de una señal para continuar.

- *Actividades complejas* (clase *ComplexActivity*. Se pueden descomponer en actividades más pequeñas. Existen los siguientes tipos de actividades complejas:
  - *Todo*. Ejecuta actividades en paralelo.
  - *Selección*. Elige una actividad, de entre un conjunto de alternativas, en función de un evento.
  - *Para todo*. Ejecuta cada una de las actividades de una lista.
  - *Secuencia*. Ejecuta actividades en secuencia.
  - *Condicional*. Ejecuta condicionalmente una actividad de entre un conjunto de ellas.
  - *Repetir hasta*. Ejecuta un conjunto de actividades una o más veces hasta que se alcanza una condición de salida.
  - *Repetir mientras*. Ejecuta un conjunto de actividades cero o más veces mientras se cumpla una determinada condición.

- *Procesos* (clase *Process*). Son actividades complejas que pueden invocarse desde otros procesos y que se componen de un conjunto de actividades (clase *Activity-Set*). Un proceso que se define independientemente de otros procesos se llama *proceso de alto nivel*, mientras que si se define para ser ejecutado dentro de otro proceso se denomina *proceso encadenado*. Además, existen otros dos tipos de procesos que tratan situaciones especiales: *excepciones* y *compensaciones*.

- *Contextos* (clase *Context*). Definen un entorno para la ejecución de actividades relacionadas entre sí: se utilizan para intercambiar información y coordinar la ejecución del WF. Un contexto contiene definiciones locales que sólo se aplican dentro de un determinado ámbito, y que pueden incluir entre otros elementos, propiedades, procesos, y señales. Los contextos pueden encadenarse de forma que el contexto de los procesos hijo puedan heredar el del padre.

- *Propiedades* (clase *Property*). Se utilizan para intercambiar información y sólo existen dentro de un contexto. Una propiedad tiene un nombre y un tipo, y sus instancias tienen un valor con un rango que se corresponde con el tipo de su definición.

- *Señales*. Se utilizan para coordinar la ejecución de las actividades dentro de un contexto común, en lugar de utilizar construcciones de control típicas como secuencias o separaciones. En BPML las señales se entienden como mensajes y algunas actividades simples se utilizan para enviar esos mensajes (actividad atómica *lanzar*) y para esperar por esos mensaje (actividad atómica *sincronizar*).

2.4.1.1. Ventajas de usar BPML

Al estar desarrollado por BPMI.org está soportado por varias organizaciones incluyendo Intalio, SAP, Sun, y Versata. Además, está fundamentado en π-calculus, con lo que es formal y, por lo tanto, no ambiguo. Ahora bien, no hemos encontrado evidencias de este hecho, ya que no se incluyen las correspondencias con π-calculus que relacionen ambas especificaciones. Otra ventaja de este estándar es que soporta directamente los principales patrones de control presentes en las herramientas comerciales de WFs.

2.4.1.2. Inconvenientes de usar BPML

BPML es un lenguaje para ser ejecutado, por lo que no tiene una notación gráfica para el diseño de procesos. Está especificado en XML Schema [33], que se caracteriza entre otros aspectos por ser un formato pensado para ejecutar procesos en entornos de sistema a sistema, es decir, sistemas que no tienen en cuenta la interacción humana y, por lo tanto, la legibilidad de los modelos. En cuanto a la expresividad, BPML captura menos patrones que los modelos orientados a la representación como son BPMN o XPDL y también es menos expresivo que otras lenguajes para ejecutar WFs, tal y como se muestra en las tablas 2.2 y 2.3. Entre las limitaciones más importantes de

BPML cabe destacar que no permite representar patrones de mezcla avanzados, bucles arbitrarios o hitos [258].

Finalmente mencionar que la expresividad de los lenguajes estructurados en bloques, como BPML o BPEL, está limitada a procesos “bien estructurados”, donde existe una correspondencia uno-a-uno entre las separaciones y las uniones; es decir, todas las separaciones se cierran con una unión, asegurando que no existe descompensación entre el número de separaciones y uniones. Esto fuerza al diseñador a introducir entidades del tipo *signal*, *raise* y *synch* para emular las características de los lenguajes basados en grafos como BPMN o UML.

2.4.2. BPEL4WS

El lenguaje BPEL [15] (del inglés, *Business Process Execution Language*), también conocido como BPEL4WS (*BPEL for Web Services*), es un lenguaje de orquestación de servicios web definido por OASIS[7]. Es la convergencia del modelo teórico basado en π-calculus de XLANG [246] con la aproximación más tradicional basada en redes de Petri de WSFL [168]. El modelo de procesos resultante es formal y permite expresar la concurrencia y el dinamismo entre las tareas sin ambigüedades. Aunque existen algunos trabajos que asocian una notación gráfica a BPEL [132, 238, 208], la especificación del estándar no define ninguna y por ello su legibilidad es menor respecto a otros lenguajes. BPEL trata el modelado de dos tipos de procesos:

- *Procesos abstractos.* Es un protocolo de negocio que especifica el intercambio de mensajes entre distintos elementos, pero sin revelar el comportamiento interno de cada uno de ellos.
- *Proceso ejecutable.* Especifica el orden de un conjunto de actividades, los participantes, los mensajes intercambiados entre dichos participantes y el mecanismo de gestión de excepciones.

La especificación de un proceso en BPEL es parecido a un diagrama de flujo. Cada elemento del proceso se llama actividad y puede ser *primitiva* o *estructurada*. Una actividad primitiva puede ser de los siguientes tipos:

- *Invocar.* Realiza una llamada a una operación de un servicio web descrito en WSDL.
- *Recibir.* Espera por un mensaje de una fuente externa.
- *Responder.* Envía un mensaje a una fuente externa.
- *Asignar.* Copia datos de un contenedor a otro.
- *Lanzar.* Indica un error en la ejecución

[7] http://www.oasis-open.org

- *Terminar* . Finaliza la instancia de ejecución del servicio.
- *Vacío*. No hace nada.

Para facilitar la representación de construcciones de control BPEL facilita el siguiente conjunto de actividades estructuradas que se pueden encadenar a través de enlaces de control llegando a formar grafos acíclicos dirigidos:

- *Secuencia*. Define el orden de ejecución.
- *Condicional*. Define estructuras condicionales.
- *Repetir mientras*. Define estructuras iterativas.
- *Seleccionar*. Define condiciones de carrera basadas en tiempo o eventos externos.
- *Flujo*. Define estructuras que permiten la ejecución de actividades paralelas.
- *Ámbito*. Agrupa actividades en un mismo gestor de errores.

2.4.2.1. Ventajas de usar BPEL

Se puede considerar BPEL como un subconjunto del lenguaje BPML. Es importante aclarar que a pesar de partir de un subconjunto del lenguaje BPML, la evolución del lenguaje BPEL hace que en la actualidad sea más expresivo. Ambos comparten las mismas tecnologías de servicios web (SOAP, WSDL) y de XML (XPath, XSDL) y han sido diseñados conjuntamente con otras especificaciones (WS-Security, WS-Transactions). Sin embargo, BPEL no soporta algunas características de BPML de gran utilidad para el modelado de WFs como son el encadenamiento de procesos (en BPML un proceso es una actividad) o las transacciones complejas. A pesar de ello, BPML está siendo abandonado con la adopción de BPEL dentro de la pila de estándares de la iniciativa *Business Process Management Initiative*[8]. Debido a: *(i)* la cada vez mayor influencia de los servicios dentro de los motores de WFs y *(ii)* que BPEL es en la actualidad el estándar de referencia para la orquestación de servicios web.

2.4.2.2. Inconvenientes de usar BPEL

Al igual que BPML, BPEL está pensado para ejecutar procesos en entornos de sistema a sistema y, por lo tanto, no tiene en cuenta la interacción humana. A pesar de ello, ha sido muy utilizado por los sistemas de WFs que tienen su origen en las soluciones de integración. Con la salida de la especificación BPEL4People [4], BPEL 2.0 [9] ha añadido soporte para la interacción basada en roles, y con ello permite asignar tareas a roles y delegar su ejecución a una persona.

[8]http://www.bpmi.org

En cuanto a la capacidad expresiva, como se puede observar en las tablas 2.2 y 2.3, BPEL representa menos patrones que los modelos orientados a la representación como son BPMN, UML o XPDL, pero más que otros lenguajes de ejecución como BPML [299]. Ello se debe a que BPEL elimina algunas restricciones heredadas por BPML de WSFL/IBM MQSeries. Sin embargo, adolece de patrones importantes como el de ciclos arbitrarios, que es un patrón muy común en los sistemas de WF.

BPEL implementa algunos de los patrones de datos descritos en el apartado 2.1.1.1. Como se puede apreciar en la Tabla 2.4, soporta menos de la mitad de estos patrones, y da un soporte similar al de los lenguajes de representación UML y XPDL, aunque inferior al de BPMN.

Finalmente, comentar que al igual que BPML o YAWL, BPEL no incorpora elementos que permitan modelar los patrones recursos descritos en el apartado 2.1.1.3. Sin embargo, algunas versiones comerciales del lenguaje, como Oracle-BPEL[9] o WebSphere-BPEL[10], sí dan soporte a la mayor parte de estos patrones[11].

2.4.3. YAWL

YAWL (del inglés *Yet Another Workflow Language*) [259] es un lenguaje creado específicamente para ejecutar los patrones de comportamiento de los WFs. Las versiones iniciales de YAWL estaban orientadas a la perspectiva de control, y permitían implementar 19 de los 20 patrones originales de comportamiento [260]. En la actualidad están trabajando en una nueva versión (*newYAWL*) [220] que añade un soporte limitado a los patrones de datos y recursos, aunque este esfuerzo está dificultado por la falta de una descripción formal de este tipo de patrones.

YAWL está fundamentado en un tipo de redes de Petri de alto nivel llamadas *WF-nets* [255], aunque extiende su modelo para dar soporte a algunos de los patrones de control más avanzados [259]. Al estar basado en redes de Petri no necesitaría de otro formalismo de representación gráfica como BPMN o los diagramas de actividad de UML. Sin embargo, para simplificar la representación de los procesos, YAWL define un lenguaje de representación propio cuyos principales elementos se muestran en la Figura 2.27. Este lenguaje mantiene el característico grafo bipartito de las redes de Petri, donde los círculos representan a las condiciones, los rectángulos a las tareas y los arcos asocian dichos elementos. Sin embargo, al contrario que las redes de Petri de alto nivel, los arcos pueden conectar a dos tareas. Esta diferencia es únicamente a efectos de representación, ya que internamente se considera que existe una condición oculta entre ambas tareas. YAWL define tres tipos de condiciones:

- *Condición de entrada.* Todo WF tiene una única condición de entrada a partir de la cual se inicia la ejecución.

[9] http://www.oracle.com/technology/products/ias/bpel/index.html
[10] http://www.redbooks.ibm.com/abstracts/sg246381.html?Open
[11] http://www.workflowpatterns.com/evaluations/standard/index.php

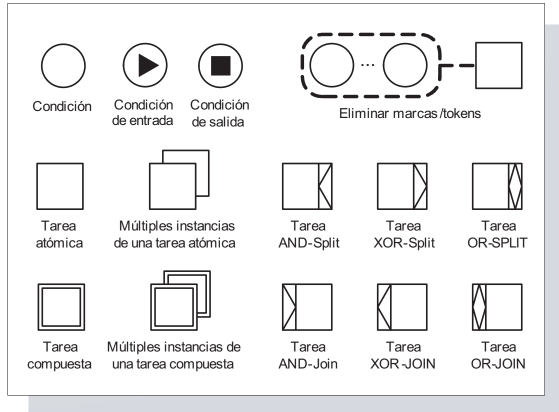


Figura 2.27: Elementos básicos de YAWL

- *Condición de salida.* Todo WF tiene una única condición de salida que indica la finalización del proceso.
- *Condición.* Permite representar las condiciones intermedias de los procesos, de modo que cada uno de estos nodos será la condición de salida de una tarea y la condición de entrada de otra.

Además, YAWL define dos tipos de tareas que pueden tener asociadas múltiples instancias:

- *Tareas atómicas.* Son tareas simples cuya resolución no se descompone en (sub)-tareas. Dentro de este tipo de tareas se han definido una serie de tareas que están asociadas con el enrutamiento del proceso: existen tres tareas de separación, *AND-SPLIT*, *XOR-SPLIT* y *OR-SPLIT*, y tres tareas de unión, *AND-JOIN*, *XOR-JOIN* y *OR-JOIN*.
- *Tareas compuestas.* Se refieren a otra red y sirven para definir un proceso de modo jerárquico: la red que no está apuntada por ninguna tarea compuesta constituirá la raíz del árbol de composición del WF.

2.4.3.1. Ventajas de usar YAWL

YAWL se creó con el objetivo de permitir la representación de la mayoría de los patrones de comportamiento identificados en los sistemas de WFs. YAWL cumple con creces este objetivo y es de largo el lenguaje con mayor expresividad. Además, es formal y proporciona un lenguaje gráfico fácil de usar y que no requiere de conocimientos de informática.

2.4.3.2. Inconvenientes de usar YAWL

Aunque la representación gráfica de YAWL está pensada para simplificar la definida por las redes de Petri, esta características es la causa de que YAWL no pueda modelar muchos de los nuevos patrones de WF definidos por la *Workflow Patterns Initiative* [302], como la mezcla de los hilos de ejecución o la unión generalizada. En [220] se propone ampliar esta notación gráfica con nuevos constructores para así mejorar la expresividad del lenguaje.

Otro de los inconvenientes de YAWL está relacionado con su implementación, ya que limita el lenguaje en algunos aspectos. Durante la fase de diseño del lenguaje, se puso de manifiesto que algunas de las extensiones que fueron añadidas a las redes de Petri eran difícil e incluso imposibles de codificar. Como resultado, la semántica formal original de YAWL está definida como un sistema de transiciones etiquetado y no en términos de redes de Petri [259].

Finalmente, comentar que al igual que BPEL y BPML, el lenguaje YAWL no incorpora elementos que permitan modelar los patrones de datos y recursos. Sin embargo, el sistema YAWL da soporte a alguno de estos patrones a través de funcionalidades ajenas al lenguaje.

2.5. Lenguajes basados en tecnología semántica

Los lenguajes vistos hasta este momento aportan un marco de definición de WFs a nivel sintáctico: A través de un lenguaje gráfico, de texto o de etiquetas definen una gramática para describir la estructura de los procesos. Si bien ésta es la aproximación tradicional seguida en la mayoría de los sistemas de WFs (y de los lenguajes de programación), en los últimos tiempos se han venido desarrollando lenguajes que permiten enriquecer este enfoque con *semántica*. Estos lenguajes se construyen sobre ontologías [122], que describen semánticamente sus componentes y permiten razonar acerca de la funcionalidad de esos componentes (por ejemplo, procesos, construcciones de control, etc.) y de cómo interactúan entre ellos. El objetivo de estos lenguajes es definir un modelo abstracto que capture la semántica del comportamiento.

La mayoría de estos lenguajes se encuadran dentro del ámbito de los servicios web semánticos (tecnología semántica + servicios web) [31], y dada la capacidad de estos lenguajes y la profunda incidencia del mundo de los servicios en la tecnología de WFs, están comenzando a ser utilizados como lenguajes de representación/ejecución en algunos sistemas de WF [25].

Los lenguajes más representativos de esta aproximación son: OWL-S [174] y WS-MO [83]. Estos lenguajes han sido propuestos al W3C como recomendaciones y ambos proporcionan capacidades de coreografía y orquestación de servicios web, con lo que pueden utilizarse para modelar WFs. En este sentido, los lenguajes de composición de servicios web se están usando cada vez más con este mismo propósito, lo cual ha venido motivado por la integración de BPEL dentro de la mayor parte de los sistemas de WFs. Además, las ontologías que describen estos lenguajes posibilitan la

automatización de ciertas funcionalidades relacionadas con el uso de los servicios [20]:

- *Publicación.* Hace disponible la descripción de la funcionalidad del servicio.
- *Descubrimiento.* Localiza diferentes servicios que son adecuados para la ejecución de una determinada tarea.
- *Selección.* Escoge el servicio más adecuado de entre un conjunto de servicios disponibles.
- *Composición.* Combina servicios para conseguir un objetivo.
- *Mediación.* Resuelve desajustes (datos, protocolo, proceso) entre servicios combinados.
- *Ejecución.* Invoca servicios.
- *Monitorización.* Controla la ejecución del proceso.
- *Compensación.* Proporciona soporte transaccional para deshacer o mitigar efectos no deseados.
- *Sustitución.* Facilita la sustitución de servicios por otros equivalentes.

Algunas de las funcionalidades anteriores, como por ejemplo, la selección o la composición, son directamente aplicables a los WFs. Otras en cambio, como la publicación o el descubrimiento, requieren de una infraestructura web fuera del ámbito de los sistemas actuales de WFs. Sin embargo, una descripción semántica sí facilitaría la creación de repositorios de WFs más ricos y reutilizables. A través de esos repositorios, la selección del WF más adecuado, la composición y mediación entre estos procesos, e incluso la ejecución y sustitución de partes de un proceso, podrían enriquecerse gracias al uso de ontologías. Es sencillo, por lo tanto, entrever las ventajas de aplicar la tecnología semántica a los WFs.

2.5.1. OWL-S

OWL-S [174] es una de las ontologías más empleadas a la hora de definir servicios web semánticos. Se dicen *semánticos* debido a que tanto su especificación, en el lenguaje de representación de ontologías OWL [84], como su estructuración permiten a los agentes razonar o inferir conocimiento acerca del servicio. Tal y como se explicó en el Capítulo 1, OWL-S describe un servicio desde tres perspectivas diferentes, aunque complementarias:

- El `ServiceProfile` responde a la cuestión *"qué hace el servicio"*, describiendo sus entradas/salidas, sus características funcionales, como son sus condiciones de aplicabilidad, y algunas características no funcionales, como es la calidad de servicio.

- El `ServiceGrounding` responde a la cuestión *“cómo puede un agente acceder al servicio”*, describiendo los detalles del protocolo de comunicaciones, formato de los mensajes y puertos de comunicación entre otros aspectos.
- El `ServiceModel` responde a la pregunta *“cómo funciona el servicio”*, detallando al cliente el contenido semántico de la solicitud, las condiciones bajo las cuales un determinado resultado puede producirse y los pasos que llevan a dicho resultado.

Cada servicio posee un conjunto de perfiles, de forma que pueda proporcionar diferentes funcionalidades a los clientes. Sin embargo, cada servicio *puede* tener un único modelo de servicio, donde indica su funcionamiento, y un único modelo de conexión (*grounding*, en inglés), donde define la comunicación a bajo nivel con el cliente. Teniendo esto en cuenta, el perfil del servicio se puede usar para el descubrimiento y la selección de servicios; mientras que el modelo de servicio y de conocimiento base se usan para describir la operacionalidad y la invocación a bajo nivel del servicio, respectivamente.

El modelo de servicio, representado en la Figura 2.28, captura la coreografía del servicio, pero al contrario que los lenguajes tradicionales, la describe a través de una ontología de procesos. La raíz de esta ontología es el concepto de *proceso* y se define a partir de sus entradas, salidas, pre- y postcondiciones, las restricciones sobre las salidas y los efectos en el entorno. El modelo del proceso se describe a través de una estructura de árbol donde los nodos internos representan procesos *compuestos* y los nodos hoja son procesos *atómicos*, que tienen asociada una conexión base y son, por lo tanto, directamente invocables. Un conjunto de construcciones permiten estructurar el control de los procesos compuestos:

- *Sequence*. Permite secuenciar la ejecución de construcciones de control.
- *Split*. Permite ejecutar un conjunto de construcciones de control de forma concurrente. La finalización de este constructor no requiere que los procesos concurrentes lleguen a concluir.
- *Choice*. Permite ejecutar una única construcción de control de un conjunto de construcciones de control seleccionables.
- *Split-Join*. Permite separar y sincronizar la ejecución de un conjunto de construcciones de control.
- *Any-Order*. Permite ejecutar un conjunto de construcciones de control de forma secuencial, aunque sin un orden predeterminado.
- *If-Then-Else*. Permite ejecutar una de las dos ramas de control en función de si se cumple o no una determinada condición.
- *Repeat-Until*. Permite repetir la ejecución de una construcción de control hasta que se cumpla una condición de salida.

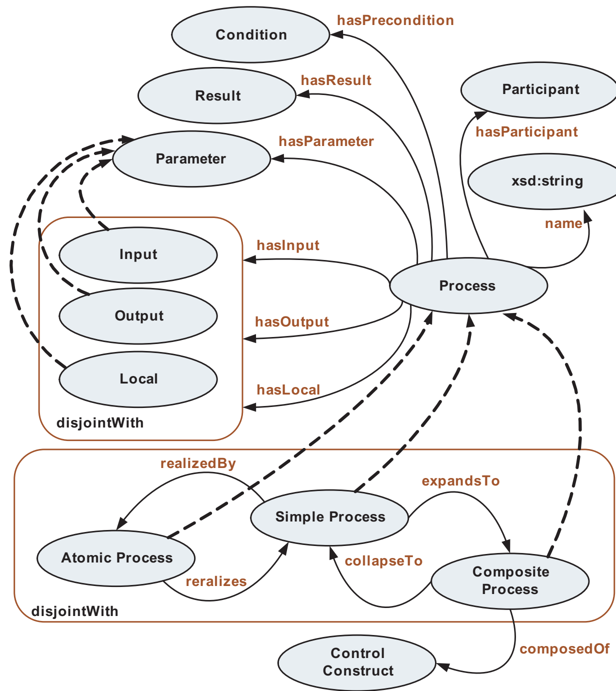


Figura 2.28: Taxonomía de OWL-S para la descripción de procesos

- *Repeat-While.* Permite repetir la ejecución de una construcción de control mientras se cumpla una condición.

- *Perform.* Representa la ejecución de un proceso. Si este proceso es atómico, implica la ejecución de una operación WSDL, mientras que si es compuesto, consiste en la ejecución de la estructura del nuevo proceso.

2.5.1.1. Ventajas de usar OWL-S

OWL-S es en sí mismo una ontología OWL [84]. Por ello, su uso para describir un proceso aporta una capa semántica que permite razonar acerca de las características de dicho proceso. Algunas de estas características también se especifican mediante otras ontologías: es el caso de los parámetros de entrada, locales y de salida, que se definen como variables de la ontología SWRL [137], o de las condiciones y efectos, que se especifican mediante fórmulas SWRL.

OWL-S define una clara separación entre la descripción del servicio y su implementación. Por ejemplo, la Figura 2.29 muestra esquemáticamente la asociación entre el modelo de servicio y el modelo de conocimiento base para asociar a los procesos

atómicos su implementación a través de una operación WSDL. El punto de encuentro entre los dos modelos es la información intercambiada: en el caso de los servicios atómicos, el conocimiento base define las correspondencias entre los parámetros especificados al nivel de la ontología y los usados por la operación WSDL. Esta separación permite razonar acerca de cómo usar el servicio, de la información que maneja y de su comportamiento *independientemente* de su implementación.

Otra de las ventajas de OWL-S es que no pretende competir o reemplazar otros estándares de descripción, publicación o comunicación de servicios web. OWL-S sólo pretende aportar una capa semántica por encima de estos estándares. Por ello, sigue utilizando estándares asentados en la web como WSDL para la descripción de servicios, SOAP para la invocación y comunicación, y UDDI para el descubrimiento de servicios [199].

2.5.1.2. Inconvenientes de usar OWL-S

Como principal problema cabe destacar la ambigüedad de su modelo de composición de procesos. En la especificación OWL-S, los constructures de control están definidos textualmente y a través de ejemplos muy sencillos. Esta forma de describir una semántica operacional es muy imprecisa y suele tener asociada problemas de interpretación. El problema de fondo es la falta de un formalismo de representación, aunque este punto ha generado un gran interés en la comunidad investigadora y existen varias propuestas para abordarlo y resolverlo [191, 269, 89, 34, 308, 64, 14, 183, 81] (estas propuestas se analizan en el Capítulo 9). OWL-S es, por lo tanto, ambiguo y puede provocar que cada motor de servicios OWL-S defina una semántica de ejecución particular. De esta forma, la ejecución de un mismo servicio en dos motores diferentes puede devolver resultados también diferentes.

Por otra parte, OWL-S no incorpora los patrones avanzados, de cancelación, o instanciación múltiple descritos en el apartado 2.1.1.1, y por ello tiene poca expresividad comparado con los demás lenguajes de WFs. En cambio, como se puede ver en las tablas 2.2 y 2.3, sí da soporte a la mayoría de patrones básicos de control utilizados en la mayoría de los motores de WFs. Una particularidad de la especificación OWL-S es que recoge construcciones de control no reconocidas como patrones de WFs, como por ejemplo, los patrones *sin orden* y *si-entonces-sino*, y otros patrones que definen un comportamiento ligeramente distinto respecto al patrón de WFs, como el patrón *separación*.

En cuanto a los patrones de datos, los parámetros de OWL-S serán globales y se pasarán por referencia. Esto significa que la visibilidad de los parámetros será siempre a nivel de caso. Este se debe a que OWL-S especifica sus parámetros en el lenguaje OWL y en éste no es posible indicar el ámbito de aplicación del parámetro. Además, la interacción es menor que la proporcionada por BPMN, UML o XPD, tal y como se muestra en la Tabla 2.4, pero es ligeramente mayor que la de BPEL debido a que OWL-S soporta (sub)flujos de trabajo. Así, permite la interacción entre tareas (una tarea se corresponde con un proceso OWL-S) y entre (sub)procesos y tareas. Los procesos OWL-S además incorporan precondiciones y postcondiciones declaradas a

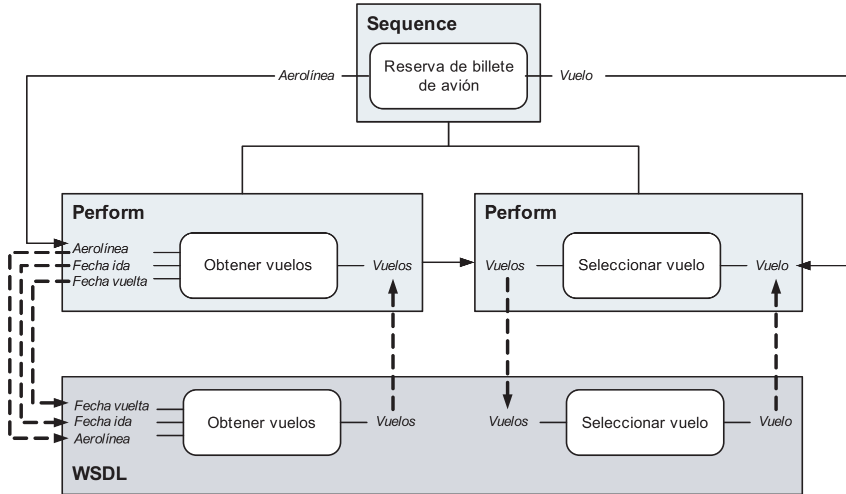


Figura 2.29: Ejemplo de integración entre el modelo de servicio y la capa de conocimiento de OWL-S

través del lenguaje SWRL [137], con lo que también soporta parte de los patrones de datos. Por último, este lenguaje soporta una gran variedad de operadores cubriendo las operaciones de existencia y valor que afectan a los datos. A pesar de ello, OWL-S no soporta por completo el enrutamiento basado en datos, ya que no soporta el patrón *elección múltiple*.

El soporte de los patrones de organización es muy reducido, ya que en OWL-S la única referencia a un componente de organización es la relación entre el proceso y el participante, como se muestra en la Figura 2.28. Por ello únicamente se puede afirmar que OWL-S soporta una asignación directa de tareas.

Finalmente, su utilización estaría restringida al ámbito de la ejecución al no disponer de una representación gráfica. Aunque esta ontología se especifica en el formato OWL, no deja de ser en último término un fichero RDF o en el mejor de los casos una representación gráfica de la jerarquía de clases de la ontología.

2.5.2. WSMO

WSMO [213, 83] es una ontología que describe los componentes principales de WSMF [103], un marco de conocimiento que proporciona un modelo conceptual para descubrir, ejecutar y componer servicios web. La novedad de WSMO es que *(i)* introduce el concepto de *mediación* como parte de la infraestructura que trata el descubrimiento y la composición de servicios web: los mediadores describen las conexiones entre los componentes de WSMF; y *(ii)* representa la orquestación y la coreografía a través de reglas de transición definidas mediante máquinas de estados abstractos (ASMs, del inglés *Abstract State Machines*) [124].

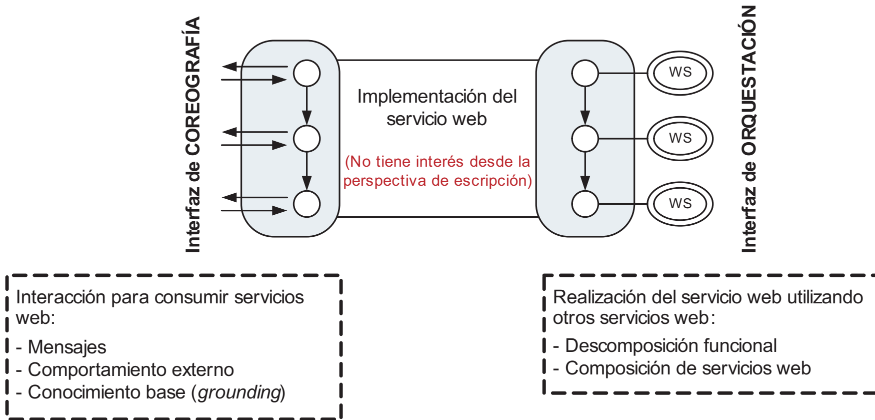


Figura 2.30: Orquestación y coreografía de WSMO

En la Figura 2.30 representamos las interfaces de coreografía y de orquestación que constituyen la base para la composición de servicios en WSMO [214]. Ambas interfaces se especifican a través de un modelo de estados inspirado en el formalismo de las ASMs que proporcionan los mecanismos básicos para modelar la interacción entre los proveedores de servicios y los clientes en el nivel abstracto, y su uso tiene varios beneficios:

- *Minimalidad.* Las ASMs tienen un pequeño conjunto de primitivas de modelado.
- *Expresividad.* Las ASMs permiten simular máquinas de Turing y pueden, por lo tanto, modelar cualquier tipo de computación.
- *Formalidad.* Las ASMs definen un marco formal para expresar sistemas dinámicos.

La coreografía y orquestación de WSMO toma prestados los mecanismos básicos de las ASMs:

- Una firma define los predicados y funciones que se usarán en la descripción.
- Los hechos especifican los estados de la base de datos.
- Los cambios de estado se describen mediante reglas de transición, que especifican cómo los estados cambian haciendo falsos (o borrando) algunos hechos previamente ciertos y haciendo ciertos (o insertando) otros hechos.

En WSMO las firmas se definen usando ontologías. Estas ontologías están expresadas en el lenguaje WSML [47] que constituye otra de las grandes aportaciones de la propuesta WSMO, y le proporciona un lenguaje formal para la descripción de los

servicios web. Los hechos forman la base de datos de estados y son instancias de los conceptos y relaciones definidas por la ontología en WSML. Los cambios de estados se describen en términos de creación de nuevas instancias o cambios en el valor de algún atributo de la ontología.

Las reglas de transición de estados utilizadas en WSMO tienen una de las siguientes formas:

- **if** Condition **then** Rules
- **forall** Variables **with** Condition **do** Rules
- **choose** Variables **with** Condition **do** Rules

La parte Condition se refiere a una condición o expresión lógica (también llamada *guarda*) definida en WSML [212]. Las parte de *Rules* es un conjunto de reglas ASM que representan cambios primitivos de estado como añadir, borrar o modificar algún hecho. Las reglas de transición tipo **if** − **then**, **forall** y **choose** pueden usarse para definir cambios de estado más complejos. Como es usual con las ASMs, las reglas de la coreografía de WSMO y los cambios en el estado se evalúan en paralelo. Cuando una regla de una ASM es ejecutada, el motor de ASMs realiza una transición de un estado de la base de datos a otro.

Una ejecución de una coreografía u orquestación en WSMO consiste, por lo tanto, en una secuencia finita o infinita de estados $s_0, s_1, \ldots$ donde s_0 es el estado inicial de la coreografía y, para cada $n \geq 0$, se puede realizar una transición desde el estado s_n al estado s_{n+1} a través de la ejecución de un conjunto de reglas que manipulan los hechos.

2.5.2.1. Ventajas de usar WSMO

No se puede considerar WSMO como un simple lenguaje, sino un metamodelo basado en ontologías (especificadas en WSML) que permite describir los aspectos más relevantes de los servicios web semánticos. Esta estructuración es uno de los puntos fuertes de WSMO para dar soporte al despliegue e interoperabilidad de servicios. Las ontologías son el núcleo de WSMO y todos sus componentes se definen a través de ellas. Por ello, el razonamiento acerca de las características funcionales y no funcionales de un proceso estaría cubierto a través de esta solución, lo cual facilitaría la composición automática de WFs.

A través de WSMO también se facilita la reutilización de componentes: al estar todos los elementos definidos a través de ontologías y los modelos del dominio separados de la definición de los servicios, la reutilización consiste en definir mediadores entre todos estos componentes.

Otra de las características a destacar de la coreografía y orquestación WSMO es su no ambigüedad gracias al uso del formalismo ASM en combinación con el lenguaje WSML. Ello permitiría usar una máquina virtual WSMO como ejecutor de WFs, sin

embargo, en la actualidad aún no se dispone de una implementación de una máquina o motor que permita ejecutar coreografías y orquestaciones WSMO.

2.5.2.2. Inconvenientes de usar WSMO

Desde el punto de vista del modelado de WFs, el principal inconveniente de la propuesta WSMO es su poca legibilidad. WSMO utiliza ASM para modelar la coreografía y orquestación de procesos, que en último término son reglas de transición. A medida que se añaden procesos y las condiciones de aplicabilidad se hacen más complejas, la legibilidad del modelo se reduce drásticamente y es sólo manejable por parte de un experto informático.

Además, es necesario mencionar que tanto la semántica de los interfaces web (coreografía y orquestación) como el conocimiento base que permite invocar servicios externos, no están completamente definidos. Por ejemplo, no está definida la relación del conocimiento base de WSMO con WSDL, el actual estándar de descripción de servicios, complicando así la aplicabilidad de esta solución a un desarrollo de WFs.

Al igual que los demás formalismos utilizados para representar procesos, las ASM permiten emular máquinas de Turing y, por lo tanto, cualquier tipo de computación. Sin embargo, la definición de algunos de los patrones de WFs resulta sumamente complicada y poco intuitiva para un diseñador inexperto. Sería necesario incluir una capa de patrones por encima de WSMO para así facilitar el uso de WSMO como lenguaje de representación de WFs. Por esta razón WSMO sale mal parado en la comparativa que se muestra en las tablas 2.2 y 2.3: las ASMs permiten sin ninguna duda dar soporte a la mayoría de estos patrones, sin embargo, no es un soporte directo con lo cual todas las entradas de la tabla tendrían un '-'. En cualquier caso, este problema también lo tienen otros formalismos como π-calculus, las ASMs o las redes de Petri, pero, en el caso de las redes de Petri está atenuado por el hecho de que son un formalismo también gráfico que con un poco de notación adicional (como es el caso del lenguaje YAWL) permite fácilmente dar soporte a la mayor parte de los patrones.

Finalmente, mencionar que a pesar de definir un metamodelo para servicios, no está claro que WSMO esté estructurado adecuadamente para maximizar la reutilización de los componentes que definen un WF. Algunas dimensiones, como la de recursos, no están claramente identificadas dentro de WSMO.

2.6. Conclusiones

El campo del modelado de WFs ha sido muy activo desde la implantación de los primeros sistemas sistemas de información de oficinas en torno a los años setenta [96, 135, 310]. Estas primeras aproximaciones estaban basadas en redes de Petri, pero desde entonces han aparecido más lenguajes con el fin de mejorar el modelado de WFs. Además de los analizados en los apartados anteriores, merece la pena mencionar FDML [129], BPSS [72], XLANG [246], WSFL [168], o WSCI [18], que en

algún momento llegaron a implantarse en alguno de los muchos sistemas que utilizan WFs: WFM (del inglés *Workflow Management*), BPM (del inglés *Business Process Management*), B2B (del inglés *Business-to-Business*), CRM (del inglés *Customer Relationship Management*) o ERP (del inglés *Enterprise Resource Planning*). Además, el cambio propiciado en la arquitectura de estos sistemas con la aparición del paradigma de servicios web ha acelerado la aparición de nuevos lenguajes. Así, varios productos comerciales como *WebSphere MQ Workflow*[12] y *Oracle BPEL Process Manager*[13] ya usan BPEL tanto para el modelado como para la ejecución de WFs.

También resultan llamativos los distintos enfoques desde los que se promueve este modelado. Algunos lenguajes modelan los WFs desde la perspectiva de la representación. Por ejemplo, BPMN extiende los diagramas de flujos para modelar diagramas de procesos, incorporando una semántica gráfica con una nueva notación más acorde con las necesidades de los WFs. Otro ejemplo son los diagramas de actividad de UML, que a pesar de no haber sido creados para modelar WFs, son ampliamente usados para diseñar WFs. Dentro de esta categoría también se pueden encontrar lenguajes pensados para el intercambio de WFs entre herramientas visuales. Éste es el caso del lenguaje XPDL que aboga por una representación que capture los aspectos comunes tanto de modelado como de representación gráfica.

En el lado opuesto están los lenguajes de ejecución de WFs. Los principales representantes de esta categoría son BPML y BPEL, que aproximan la ejecución desde la orquestación de servicios. Estos lenguajes están basados en algún formalismo de procesos para evitar ambigüedades y así no permitir que distintas implementaciones puedan interpretar y ejecutar al mismo WF de manera distinta. Sin embargo, no suelen tener representación gráfica, por lo que se combinan con los lenguajes de representación. En estos casos surge el problema de expresividad, ya que no suele existir una correspondencia directa entre los elementos del lenguaje de representación y los del lenguaje de ejecución. Una excepción es YAWL, que además de semántica de ejecución fundamentada en un modelo formal, también proporciona un lenguaje gráfico.

Dentro de los formalismos de representación de procesos destacan las redes de Petri, π-calculus y las ASMs. El primero es de largo el más extendido para el modelado de este tipo de procesos: su capacidad para modelar la concurrencia unida al hecho de que es un formalismo gráfico son el principal motivo del éxito de estas redes. En este sentido, existen muchos ejemplos del uso de redes de Petri en sistemas de WFs tanto de investigación como comerciales. Algunos de estos ejemplos utilizan directamente el formalismo de redes de Petri, mientras que otros usan un lenguaje cuya semántica está basada en estas redes, como WSFL o YAWL.

En cambio, π-calculus y las ASMs no suelen usarse directamente en ninguna herramienta orientada al modelado de WFs. El motivo es su poca legibilidad, ya que su sintaxis es similar a la de un lenguaje de programación declarativo. Por ello, habitualmente se combinan con otro lenguaje de representación al que le añaden un soporte formal, como en el caso de BPML y XLANG, cuya ejecución está fundamentada en

[12]http://www-01.ibm.com/software/integration/wmqwf/
[13]http://www.oracle.com/technology/products/ias/bpel/index.html

π-calculus, o de BPMN en el caso de las ASMs.

Tanto los lenguajes de representación como de ejecución de WFs aportan las funcionalidades necesarias para que los sistemas tradicionales puedan modelar sus WFs. Permiten estructurar procesos a través de construcciones de control, definir sus características funcionales (entradas, salidas y condiciones) a través de un álgebra/firma, e incluso definir canales de comunicaciones mediante los cuales distintos procesos pueden comunicarse entre sí. El principal problema de estos lenguajes es la *carencia de semántica*. Las construcciones de control, los parámetros y las expresiones utilizados para caracterizar a los procesos están definidas a nivel sintáctico y por ello no es posible razonar acerca de sus características. Teniendo en cuenta que las ontologías permiten la descripción semántica de cualquier dominio, la incorporación de ontologías al modelado de procesos daría lugar a *WFs semánticos*.

Dotar de semántica a los WFs permitiría la composición y reutilización dinámica de WFs, problemas que han sido siempre difíciles de resolver para los sistemas de WFs tradicionales y que han requerido de la intervención de humanos. Con la aparición del paradigma de los servicios web semánticos, esta tecnología podría aplicarse a los WFs para automatizar la composición, invocación e interoperabilidad entre procesos. OWL-S y WSMO son las principales iniciativas de la web semántica para el modelado de servicios web y, además de permitir razonar sobre las ontologías del dominio de la aplicación, permiten razonar sobre la propia estructura del proceso, ya que se definen a su vez como ontologías. Sin embargo, carecen de la expresividad característica de los lenguajes de modelado de WFs.

Redes de Petri como formalismo de representación. Del análisis llevado a cabo a lo largo de este capítulo y del uso típico de los sistemas de WFs se desprenden dos conclusiones:

1. Un **modelo gráfico** es imprescindible para garantizar la legibilidad del modelo de WFs, y por ello todos los sistemas comerciales de WFs lo incorporan. A través de algún tipo de diagrama gráfico, como UML, BPMN o redes de Petri, se facilita la comprensión de los modelos, y con ello no se circunscribe el proceso de diseño a un experto informático.

2. Un **modelo formal** es imprescindible para garantizar una ejecución sin ambigüedades. Los principales lenguajes de ejecución, como BPEL, BPML o YAWL fundamentan su comportamiento en algún formalismo.

No existen muchos lenguajes que cumplan los requisitos anteriores. Aunque la combinación de un lenguaje gráfico con uno de ejecución podría llegar a satisfacer estas condiciones, no parece ser el paso más adecuado: requiere definir un conjunto de correspondencias entre ambos lenguajes y restringir la expresividad de alguno de ellos. Por ello, hemos optado por aproximar el modelado de WFs a través de un único lenguaje, seleccionando el formalismo de las redes de Petri de alto nivel, y más concretamente el lenguaje gráfico definido para representar este tipo de redes. Aunque

existen otros lenguajes como YAWL que también cumplen con los requisitos anteriores, la principal razón por la que se optó por las redes de Petri se debe a que son el formalismo más aceptado y empleado, tienen una gran expresividad, y además poseen un importante conjunto de técnicas de simulación, validación y verificación. YAWL era otra de las posibles opciones, sin embargo, la carencia de técnicas de análisis, validación y simulación, unido a los problemas para extender YAWL y así dar soporte a muchos de los nuevos patrones de WFs [220] han motivado que optásemos por las redes de Petri.

Como se ha señalado en el Capítulo 1, los marcos conceptuales basados en conocimiento tienen problemas para el modelado de WFs por su falta de expresividad a la hora de especificar el control entre los (sub)procesos. Por este motivo, si se completa la aproximación basada en conocimiento con el formalismo de las redes de Petri estamos en disposición de definir el marco de conocimiento objeto con el que se resuelven los problemas indicados en la Tabla 1.1. una ontología de redes de Petri Para facilitar el tratamiento de estas redes como una pieza de conocimiento dentro del *metamodelo* descrito en esta memoria en esta tesis doctoral es necesario construir una ontología de redes de Petri. Esta ontología se describirá de forma detallada en el Capítulo 3.

3

Una ontología de redes de Petri para modelar procesos

La gran capacidad expresiva y de legibilidad de las redes de Petri (PNs, del inglés *Petri Net*) han convertido a este formalismo en el más empleado para el modelado de flujos de trabajo (WFs, del inglés *Workflows)*. Sin embargo, muy pocas investigaciones han profundizado en un metamodelo que permita conceptualizar y explicitar estas redes de forma que puedan ser reutilizadas con más facilidad, por ejemplo entre distintos sistemas de WFs. Si este metamodelo permitiese además modelar la dinámica de estado y ejecución de estas redes, distintos sistemas de WFs podrían compartir, intercambiar o balancear la ejecución de WFs. Este capítulo está dedicado a la descripción de una ontología de PNs de alto nivel (HLPNs, del inglés *High-Level Petri Nets*) [266] que permite asumir este reto, ya que modela tanto los grafos como la semántica de ejecución de estas redes. Además, también se presentará una ontología de redes jerárquicas, construida sobre la base proporcionada por las HLPNs, a través de la cual se definirá la semántica de composición de las HLPNs de forma que un modelo complejo pueda construirse a partir de redes más simples.

3.1. Redes de Petri

Carl Adam Petri [203] introdujo las PNs en su tesis doctoral como una herramienta para simular las propiedades dinámicas de los sistemas complejos mediante modelos gráficos de procesos concurrentes. Desde entonces su estudio y desarrollo ha tenido un auge importante debido fundamentalmente a las numerosas aplicaciones que se les han encontrado: modelos de redes abstractas, procesamiento paralelo y distribuido, teoría de grafos, problemas de transporte, problemas de decisión y reconocimiento de patrones, entre otras.

En esta sección se introducirán los conceptos básicos de las PNs [203, 190] a partir de un ejemplo intuitivo que se describe a continuación y que permitirá explicitar de forma natural las principales características de las PNs: principio de localidad, concurrencia, representación gráfica y anotación algebraica. En este caso, se escogió un ejemplo que se encuentra en el ámbito de la fabricación donde varias piezas a ensamblar llegan a una máquina de una cadena de montaje. Las condiciones y accio-

nes de este ejemplo se listan a continuación, aunque por simplicidad se restringió la fabricación/ensamblado a dos piezas:

- Lista de condiciones:
  - p1 : pieza *a* esperando
  - p2 : pieza *a* preparada
  - p3 : máquina preparada
  - p4 : pieza *b* esperando
  - p5 : pieza *b* preparada
  - p6 : pieza montada
- Lista de acciones:
  - t1 : preparar pieza *a*
  - t2 : preparar pieza *b*
  - t3 : procesar piezas

3.1.1. Redes de Petri de bajo nivel

La separación entre elementos pasivos (como son las *condiciones*) y elementos activos (como son las *acciones*) es un paso muy importante en el diseño de cualquier sistema. Esta separación está claramente soportada por el *principio de dualidad* de las PNs, que define dos conjuntos de elementos disjuntos llamados respectivamente P y T. Las entidades del mundo real que son interpretadas como elementos pasivos, se representarán como elementos tipo P (plazas, condiciones, recursos, canales de comunicación, etc.); en cambio, sí son interpretadas como elementos activos se representarán mediante elementos tipo T (transiciones, eventos, acciones, transmisión de un mensaje, etc.). Es importante mencionar que la clasificación de un objeto como activo o pasivo dependerá del *contexto*. Por ejemplo, una sentencia de un lenguaje de programación podría modelarse como un elemento activo en el contexto de una ejecución o pasivo en el contexto de una operación de compilación.

Otro de los principios esenciales de este tipo de redes es la *localidad* de las acciones. Supóngase que el estado inicial del ejemplo de fabricación previamente descrito viene dado por $m1 : [p_1 = 1, p_2 = 0, p_3 = 1, p_4 = 1, p_5 = 0, p_6 = 0]$, donde se representa la verificación de las condiciones por medio de valores binarios. Asumiendo que las acciones t_1 y t_2 requieran únicamente que las piezas estén disponibles, tanto t_1 como t_2 podrían ejecutarse en el estado $m1$. Siguiendo con el ejemplo, la ejecución de la acción t_1 podría implicar que una pieza de tipo a está prepara para ser procesada. Por lo tanto, dicha pieza pasaría de “estar esperando” a “estar preparada”. Lo mismo podría ocurrir con la ejecución de la acción t_2, pero en este caso para la pieza de tipo b. La Figura 3.1 muestra el estado $m2$ resultante de la ejecución de estas dos transiciones.

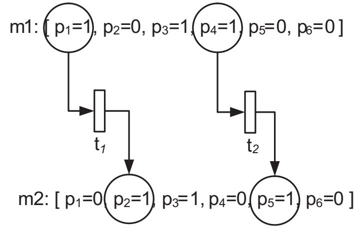


Figura 3.1: Principio de localidad y concurrencia de las acciones t_1 e t_2

La principal observación del ejemplo de la Figura 3.1 es que únicamente las condiciones relacionadas con las acciones ejecutadas se ven afectadas. Esta propiedad se conoce por el *principio de localidad* de las PNs, e indica que el comportamiento de una acción únicamente depende de su conjunto de objetos de entrada y salida. En el ejemplo anterior, la localidad de la acción t_1 es $\{t_1, p_1, p_2\}$ mientras que la de la acción t_2 es $\{t_1, p_4, p_5\}$. Se puede verificar, por lo tanto, que las localidades de las acciones t_1, y t_2 del ejemplo de fabricación no comparten condiciones. Esta propiedad define el *principio de concurrencia*, que establece que las acciones tienen que tener localidades disjuntas para poder ocurrir de forma independiente (concurrente). Es importante mencionar que la noción de concurrencia es diferente de la de paralelismo: las acciones paralelas implican una sincronización, mientras que las acciones concurrentes no están relacionadas por ninguna causalidad.

La Figura 3.1 también representa gráficamente las acciones t_1 y t_2 y las relaciona con sus precondiciones y postcondiciones mediante arcos. Los arcos conectan cada uno de los elementos de tipo T con su localidad, la cual es un conjunto de elementos de tipo P. En este caso, las condiciones (elementos de tipo P) se representan mediante círculos llamados *plazas*, mientras que las acciones (elementos de tipo T) lo hacen mediante rectángulos llamados *transiciones*. Por lo tanto, una red está constituida por un conjunto de plazas, transiciones y arcos. La Figura 3.2 representa la red completa del ejemplo de fabricación en el estado inicial m_1.

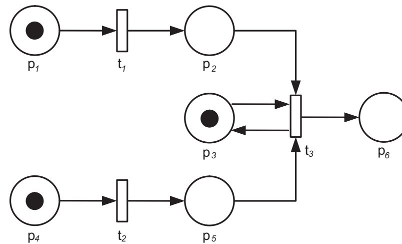


Figura 3.2: Representación del ejemplo de fabricación mediante un grafo de PN

Las PNs también pueden describirse de forma algebraica de modo que para cada representación gráfica existe una representación algebraica equivalente. Esta descripción es muy útil para el análisis matemático de estas redes.

Definición 3.1.1 *Una PN es una tripla $N = (P, T, F)$ donde:*

- *P es el conjunto de plazas,*
- *T es el conjunto de transiciones, disjunto de P, y*
- *F es el conjunto de arcos entre plazas y transiciones y establece la relación $F \subseteq (P \times T) \cup (T \times P)$.*

Si P y T son conjuntos finitos, la red N también es finita.

La definición anterior, proporcionada por Carl Adam Petri en [203], es la base de todos los modelos de PNs existentes. Por ejemplo, la red de la Figura 3.2 se puede expresar algebraicamente como: $P = \{p_1, \ldots, p_6\}$, $T = \{t_1, t_2, t_3\}$ y $F = \{(p_1, t_1), (p_2, t_2), \ldots\}$.

La verificación de una condición se representa mediante una marca (*token*, en inglés) en la correspondiente plaza. Por ejemplo, la PN de la Figura 3.2 está en el estado m_1 previamente descrito y, por lo tanto, contiene marcas en las plazas p_1, p_3 y p_4. El cambio de estado de la red se realiza por medio del disparo de alguna transición. Por ejemplo, la Figura 3.3 representa gráficamente la obtención del nuevo estado de la red cuando se disparan las transiciones t_1 y t_2. Se puede decir que una transición *puede ocurrir* o *está activa* si se verifican todas sus precondiciones, es decir, si sus plazas de entrada están marcadas. Cuando una transición ocurre, las marcas se eliminan de las precondiciones (plazas de entrada) y se crean nuevas marcas en las postcondiciones (plazas de salida). A este proceso de eliminación de las marcas en las precondiciones y de creación de nuevas marcas en las postcondiciones se le denomina *regla de transición*.

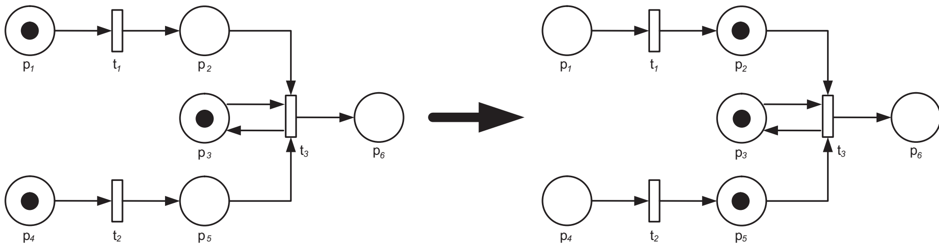


Figura 3.3: Disparo de las transiciones t_1 y t_2

Para más información acerca de las PNs consultar [190].

3.1.2. Redes de Petri de alto nivel

La PN presentada en el apartado anterior (Definición 3.1.1) describe una PN de bajo nivel. En este tipo de redes, una marca en una plaza indica la satisfacción de la condición asociada a ella. Sin embargo, dos marcas en una misma plaza son

indistinguibles. Éste es el motivo por el cual para distinguir entre las piezas de tipo a y de tipo b es necesario situarlas en dos plazas diferentes, en este caso en p_1 y p_4. Debido a esta explosión de nodos, el uso de redes de bajo nivel en sistemas complejos está desaconsejado. Las HLPNs introducen el concepto de *color* para distinguir las marcas asociadas a una plaza y con ello reducir el número de nodos y arcos de la red. La Figura 3.4 hace uso de esta propiedad para fundir (*place folding*, en inglés) las plazas p_1 y p_4 y las plazas p_2 y p_5 y definir una red equivalente.

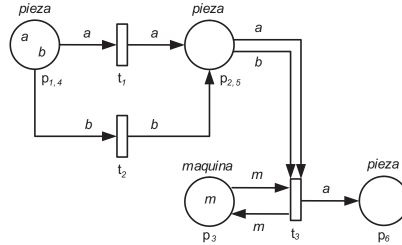


Figura 3.4: PN coloreada del ejemplo de fabricación

Cuando las marcas se pueden distinguir entonces se dice que esas marcas están *coloreadas* [149], y se clasifican por su color (tipo). En el caso de la Figura 3.4, existen dos colores: piezas=a, b y máquina=m. Se acostumbra a representar estos colores en cursiva en la parte superior de la plaza.

Al añadir color a las marcas, también es necesario indicar qué marca se debe eliminar en el disparo de una transición. Esto se hace anotando los arcos con el identificador de la marca que será eliminada. Como se puede apreciar en la Figura 3.4, la transición t_1 puede ocurrir si se verifica que la plaza $p_{1,4}$ contiene el objeto a. Además, las anotaciones asociadas a los arcos de una PN coloreada también pueden fundirse (*arc folding*, en inglés) [162] y referirse a multiconjuntos. Por ejemplo, en la Figura 3.5 los arcos entre las plazas $p_{2,5}$ y la transición t_3 de la Figura 3.4 se funden y se añade la anotación $a + b$, que representa un multiconjunto con el elemento a y el elemento b. En este caso, del disparo de la transición t_3 mantendría la misma semántica, ya que eliminaría una pieza a, otra b de la plaza $p_{2,5}$, la máquina m de la plaza p_3 y crearía una nueva marca en la plaza p_6.

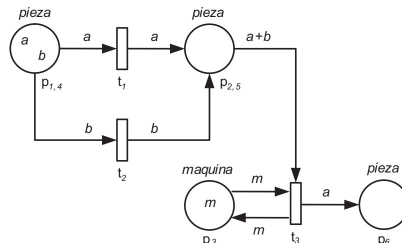


Figura 3.5: Arcos anotados mediante multiconjuntos

Esta combinación de marcas coloreadas y expresiones en arcos posibilita que las transiciones también se puedan fundir (*transition folding*, en inglés) para simplificar el modelo. Éste es el caso de la Figura 3.6 donde las transiciones t_1 y t_2 se fusionan para dar lugar a una nueva transición llamada $t_{1,2}$. Para ello, también se funden los arcos de ambas transiciones y se anotan con un multiconjunto de piezas de tipo a y b.

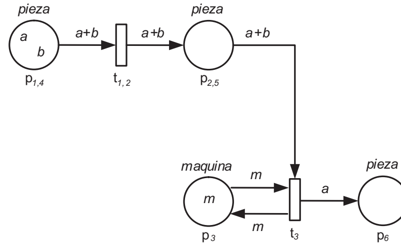


Figura 3.6: Las transicións t_1 y t_2 se funden en la transición $t_{1,2}$

Es importante mencionar que las HLPNs no tienen por qué estar anotadas únicamente con constantes, como es el caso de los ejemplos anteriores. Para hacer más extensible el modelo los arcos se anotan mediante expresiones complejas [144], que pueden estar formadas por variables y llamadas a operadores. Por ejemplo, la Figura 3.7 anota los arcos entre las plazas $p_{1,4}$ y $p_{2,5}$ con las variables x e y de tipo pieza. Con esta anotación el modelo permite ensamblar dos piezas de cualquier tipo.

Por otra parte, y volviendo al caso de fabricación previamente descrito, una máquina únicamente acepta piezas de tipo a o b. Para dar soporte a este tipo de restricciones, las HLPNs también permiten anotar las transiciones con expresiones lógicas. Estas expresiones actúan a modo de precondición y no permiten la activación de la transición si no se verifican. Por ejemplo, ahora la transición t_1 está anotada con la precondición $(x = a \wedge y = b) \vee (x = b \wedge y = a)$. Este hecho modifica el comportamiento genérico de la regla de disparo de transiciones, ya que en el caso de las HLPN será también necesario comprobar la precondición de la transición.

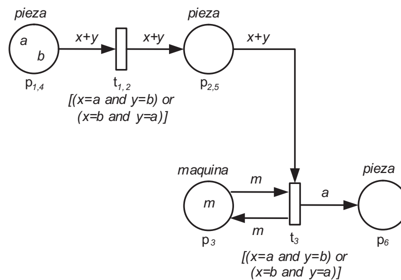


Figura 3.7: Términos anotan arcos y transiciones

La descripción algebraica de una HLPN complementa la Definición 3.1.1 con los

nuevos conceptos introducidos en este apartado.

Definición 3.1.2 *Una HLPN es una estructura* $HN = (NG, Sig, V, H, Type, AN)$ *donde:*

- NG *es el concepto de red* $N = (P, T, F)$ *descrito en la Definición 3.1.1,*
- $Sig = (S, O)$ *es una firma lógica del conjunto de tipos y operadores sintácticos,*
- V *es el conjunto de variables con tipo* S*. Este conjunto es disjunto del conjunto de operadores* O *de la firma* Sig*,*
- $H = (SH, OH)$ *es un álgebra para la firma* Sig*, donde cada tipo sintáctico* S *de* Sig *tiene asociado un tipo real de* SH *y donde cada operador sintáctico de* O *tiene asociado una función de* OH*,*
- $Type : P \to SH$ *es una función que asocia un tipo a una plaza, y*
- AN *define las funciones para la anotación de los arcos y de las transiciones.*

Lo más destacado de la Definición 3.1.2 son los conceptos de *firma* y *álgebra* [211]. Mientras la firma es una representación sintáctica de los tipos y operadores que se pueden usar para anotar la red, el álgebra aporta una representación semántica a esa representación. Por ejemplo, un operador tendrá definido su nombre, rango y dominio en el contexto de la firma, mientras que en el álgebra será una función que, además de estas propiedades, tendrá asociado un código/expresión. Por lo tanto, las funciones podrán ejecutarse y devolverán un resultado. En cierto sentido, la separación entre álgebras y firmas permite reutilizar una red sin cambiar su anotación. Por ejemplo, se puede crear una red que realiza la suma de dos números y aplicar la misma red con un álgebra que sume números enteros y otra que sume números en punto flotante.

Para más información acerca de las HLPNs consultar [144, 118].

3.1.3. Redes de Petri de jerárquicas

La creación de modelos complejos a través de HLPNs puede llegar a ser una tarea pesada. Sin embargo, de forma similar a la programación modular, el modelado de HLPNs se puede abordar mediante la composición de modelos más simples [120]. Existen muchas propuestas en la literatura que tratan distintas formas de componer un modelo de PNs [120]. Estas contribuciones han dado lugar a una gran diversidad de tipos de PNs, y la unificación de los numerosos dialectos sigue siendo en la actualidad un importante tópico de investigación [95]. Este trabajo adopta una versión simplificada de la definición de HLPN jerárquicas [149], que facilita su construcción a través de dos mecanismos de composición:

1. *Sustitución.* Este mecanismo dota de modularidad a la descripción de los sistemas e indica que un nodo de la red será sustituido por otra PN (denominada

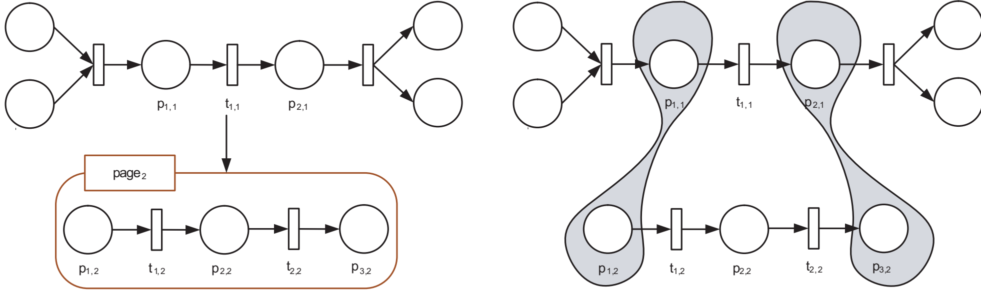


Figura 3.8: Ejemplo de sustitución (izquierda) y de fusión (derecha)

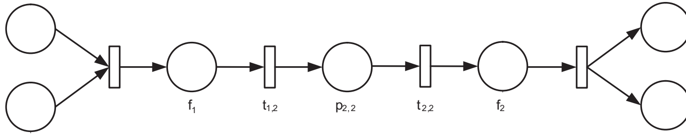


Figura 3.9: Ejemplo de red aplanada

página). En la parte izquierda de la Figura 3.8 se muestra un ejemplo donde la transición $t_{1,1}$ es sustituida por la página $page_2$.

2. *Fusión*. Este mecanismo permite especificar que un conjunto de nodos, llamados conjunto de fusión, representan un único nodo. Éste es un mecanismo muy utilizado para facilitar la representación gráfica de redes muy grandes, ya que permite representar nodos en distintas páginas y relacionarlos a través de la fusión.

La definición anterior de sustitución es muy intuitiva aunque incompleta. El mecanismo de sustitución además de identificar el nodo a sustituir y la página sustituta, también contempla un conjunto de fusiones entre ambas redes. En la parte derecha de la La Figura 3.8 se muestran las correspondencias en forma de fusiones. Puede apreciarse como las entradas y salidas de la transición $t_{1,1}$ se asocian con plazas de la página $page_2$.

Definición 3.1.3 *Una HLPN jerárquica es una estructura* $HHN = (S,\ SN,\ SP,\ SF,\ FS)$ *donde:*

- S *es un conjunto finito de páginas tales que:*
  - *Cada página* $s \in S$ *es una página HLPN no jerárquica:* $HN_s = (NG_s, Sig_s, V_s, H_s, Type_s, AN_s)$, $NG_s = (P_s, T_s, A_s)$
  - *El conjunto de elementos de cada una de estas redes es disjunto:* $\forall s_1, s_2 \in S : s_1 \neq s_2 \Rightarrow (P_{s_1} \cup T_{s_1} \cup A_{s_1}) \cap (P_{s_2} \cup T_{s_2} \cup A_{s_2}) = \phi$
- $SN \in T$, $T = \bigcup_{s \in S} T_s$ *es un conjunto finito de sustituciones de nodos.*

- *SP es una función que asigna una página a una sustitución.*
- *SF es una función que asigna una fusión a una sustitución.*
- *$FS \in P_s$ es un conjunto finito de conjuntos de fusiones donde todos los miembros de una fusión tienen el mismo color.*

La definición 3.1.3 es una versión simplificada, ya que no incluye algunos de los axiomas aplicables a estas redes. Para más información acerca de las HLPNs jerárquicas consultar [149].

Finalmente, la Figura 3.9 muestra la representación aplanada de la composición mostrada en la Figura 3.8. Como se puede ver, *(i)* la transición sustituida ha desaparecido de la red y en su lugar está la página sustituta. Además, *(ii)* cada conjunto de nodos fusionado ha sido sustituido por un único nodo. Es importante mencionar que siempre es posible trasladar la representación jerárquica a HLPN y ésta a su vez a PN. Esto significa que el poder teórico de estas tres clases de redes es el mismo. Es más, la HLPN jerárquicas no tienen una semántica operacional propia, ya que la idea subyacente es trasladar su definición a una HLPN equivalente y realizar la ejecución sobre la red aplanada.

3.2. Lenguajes de intercambio de redes de Petri de alto nivel

La construcción de una ontología es un proceso complejo y costoso. Por ello es necesario justificar su necesidad antes de comenzar tan arduo proceso y más si ya existe un estándar de intercambio de ficheros. En este apartado se compara la aproximación ontológica seguida en esta tesis doctoral con la solución propuesta por el estándar ISO/IEC 15909-2 [145] también conocido con el nombre de PNML (del inglés *Petri Net Modelling Language*) [292]. Es importante mencionar que esta comparación únicamente se refiere a la representación de HLPNs, ya que PNML está diseñado para dar soporte al intercambio de cualquier tipo y versión de PN. Por ello, el análisis se centró en la comparación de los metamodelos y en sus soluciones tecnológicas (F-Logic/OWL y RELAX NG/XML).

3.2.1. Desde el punto de vista del modelado

Nuestra ontología y el estándar de intercambio de ficheros parten de objetivos distintos. La ontología trata de modelar las HLPNs, capturando todos los aspectos de estas redes y trasladando su especificación matemática a una formulación entendible por las máquinas. En cambio, el modelo de PNML está orientado al intercambio de PNs de cualquier tipo. Por ello, PNML contiene conceptos genéricos para que de esa forma puedan ser compatibles con cualquier tipo de PN. Es más, PNML está en muy ligado al proceso de intercambio y utiliza conceptos directamente relacionados con los editores gráficos que soportan dichas redes. Por ejemplo, conceptos relacionados con aspectos gráficos o modulares también están recogidos en este estándar.

La definición del grafo anotado de la HLPN es muy similar en ambos modelos. La principal diferencia entre ambas especificaciones es que PNML se queda en el nivel sintáctico de anotación de la red. Debido a que la mayor parte de los editores no dan soporte a la implementación de las funciones usadas para anotar las redes, PNML no entra en este aspecto. En cambio, el modelo de la ontología mantiene la separación entre la firma sintáctica y el álgebra del estándar ISO/IEC 15909-1. De esta forma, se enriquece el intercambio de HLPNs con respecto a la propuesta de PNML. Al disponer del álgebra, los editores podrían hacerse más ricos y permitir ejecutar y simular HLPNs directamente a partir de la ontología intercambiada. Además, ello facilitaría la reutilización de las redes y de su álgebra. Por ejemplo, una HLPN que se encarga de gestionar una planta de fabricación podría usarse en distintos dominios simplemente cambiando el álgebra (ontología del dominio) asociada a dicha red.

La ejecución de las HLPNs no está modelada en PNML. Ésta es la principal diferencia entre ambas propuestas desde la perspectiva del modelado. Aunque capturar la semántica de ejecución no es uno de los objetivos actuales de PNML, es uno de los aspectos principales de la ontología de HLPNs. Hacer explícitos los modelos de estado y ejecución permite usar estas redes en dominios donde es necesario razonar acerca del comportamiento de la red. Por ejemplo, para comprobar el funcionamiento de servicios Grid o Web.

Ambas soluciones proporcionan un conjunto de axiomas que restringen las instancias de la taxonomía/metamodelo. Aunque PNML utiliza sentencias OCL para este propósito, dichas sentencias se restringen a la definición del grafo y aspectos como la anotación de la red no son tenidos en cuenta. Por ejemplo, PNML no verifica que las condiciones de guarda de una transición tengan un valor lógico ni que los parámetros de las llamadas a operadores tengan el tipo adecuado. En cambio, nuestra ontología detalla un conjunto de axiomas para completar la semántica de cada uno de los elementos de una HLPN.

3.2.2. Desde el punto de vista tecnológico

PNML describe un metamodelo en UML y lo traslada a un lenguaje basado en el formato XML. Las primeras versiones se basan en el lenguaje RELAX NG [73] mientras que se prevé que futuras versiones lo hagan en el lenguaje XML Schema [248]. El principal problema de la solución PNML (independientemente del uso de RELAX NG o XML Schema) consiste en que no extiende la semántica de representación proporcionada por el lenguaje XML y por ello no es lo suficientemente expresivo para describir la semántica asociada a los elementos del metamodelo. Aunque a través de esta estrategia PNML garantiza que los ficheros XML intercambiados tengan una sintaxis correcta, la semántica asociada a los elementos intercambiados debe de ser validada. Así se se hace necesario disponer de un validador del metamodelo que permita cotejar los ficheros XML intercambiados. Si el intercambio se produce en un entorno distribuido, la operación se complica, ya que además se requiere que tanto el remitente como el receptor compartan la misma *versión* del validador. La aproximación ontológica es bastante diferente respecto a este último punto: los ficheros intercambiados contienen

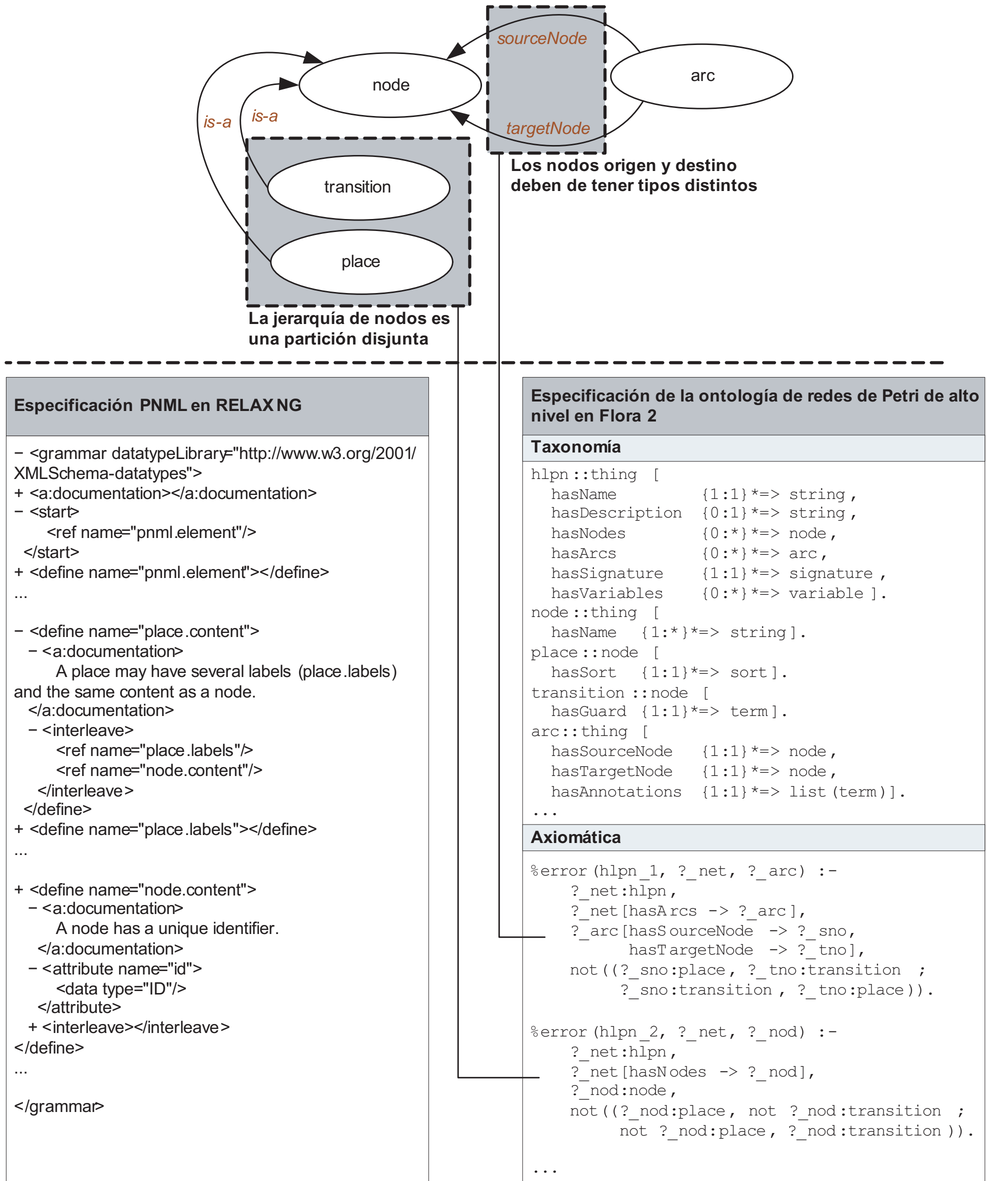


Figura 3.10: Definición del concepto nodo en las especificaciones RELAX NG de PNML y FLORA-2 de la ontología.

toda la semántica y la validación se realiza mediante razonadores estándar. En este contexto, las principales limitaciones de aplicar la estrategia seguida por PNML son:

- *No permite definir explícitamente las relaciones jerárquicas (is-a) entre dos o*

más conceptos. Al no existir mecanismos de herencia difícilmente se pueden representar las taxonomías. Por ejemplo, en los ficheros XML intercambiados por PNML las plazas y las transiciones no heredan las propiedades ni las relaciones del concepto nodo a pesar de que el metamodelo sí las contempla. La Figura 3.10 compara la especificación del concepto nodo tanto en RELAX NG como en F-Logic. En esta comparación se puede verificar como el fichero en formato RELAX NG sólo contiene los elementos XML del metamodelo mientras que la ontología en FLORA-2 además de la taxonomía contiene los axiomas que restringen los conceptos de dicha taxonomía. Además, el formato RELAX NG no permite definir relaciones jerárquicas entre los elementos y, por ello, los atributos del nodo se añaden a las plazas y transiciones. En cambio, en el formalismo F-Logic dichas relaciones jerárquicas se establecen a través del constructor :: y por lo tanto, las plazas y transiciones heredan automáticamente las propiedades y relaciones del nodo.

- *No permite definir las propiedades de las relaciones.* RELAX NG no dispone de primitivas que permitan especificar propiedades matemáticas (como la simetría o la transitividad) ni taxonómicas (como la disyunción o las particiones exhaustivas) sobre las relaciones. Estas restricciones no tienen cabida en el formato RELAX NG a pesar de que en el metamodelo de PNML alguna de ellas se exprese a través de OCL. Por ello, con la estrategia PNML es necesario asegurar que ambos lados de la comunicación disponen de las mismas restricciones a nivel del metamodelo. En cambio, en F-Logic estas restricciones se expresan bien a través de anotaciones en los conceptos bien a través de axiomas y se incorporan en el fichero intercambiado. De esta forma, ambos lados de la comunicación tienen las mismas restricciones.
- *No permite definir restricciones entre conceptos, atributos o relaciones.* Los axiomas completan la semántica de los conceptos cuando ésta no se puede expresar directamente en el modelo. Por ejemplo, el axioma *un arco conecta nodos de distinto tipo* no se puede especificar en el lenguaje RELAX NG. Por el contrario, sí puede serlo en lenguajes de ontologías como Ontolingua [122], OWL [84] o F-Logic [152]

Mencionar que estos inconvenientes no son carencias de la especificación PNML. Simplemente muestran las diferencias existentes entre la aproximación tecnológica seguida por PNML y una basada en ontologías. La propuesta de esta tesis doctoral intenta modelar el estándar ISO/IEC 15909-1 y consecuentemente asume su especificación matemática y las restricciones que dicha especificación impone tanto en la definición como en la ejecución de las redes. Por lo tanto, la solución tecnológica también debe de soportar dichas restricciones. Es más, debe de asegurar que la red transferida es correcta. Por el contrario, como se puede apreciar en la Figura 3.11, PNML no puede asegurar a través del formato de intercambio de ficheros que la red transferida es correcta, por ejemplo, no puede comprobar que un arco tiene un nodo origen y destino que son de distinto tipo, ya que ambos están definidos como atributos `IDREF` en RELAX NG. De ahí que PNML necesite acceder al nivel de la aplicación para comprobar este tipo de restricciones.

```
<pnml xmlns="http://www.informatik.hu-berlin.de/top/pnml/ptNetb">
  <net id="example1" type="http://www.informatik.hu-berlin.de/top/pntd/ptNetb">
  <name>
    <text>incorrect representation of a Petri net graph structure</text>
  </name>
  <Place id="p1">
    <name>
        <graphics><offset x="0" y="22" /></graphics>
        <text>p1</text>
    </name>
    <initialMarking>
        <graphics><offset x="-20" y="10" /></graphics>
        <text>1</text>
    </initialMarking>
    <graphics><position x="50" y="150" /></graphics>
  </Place>
  <Place id="p2">
    <name>
        <graphics><offset x="-1" y="20" /></graphics>
        <text>p2</text>
    </name>
    <initialMarking>
        <graphics><offset x="24" y="5" /></graphics>
        <text>0</text>
    </initialMarking>
    <graphics><position x="250" y="150" /></graphics>
  </Place>
  <arc id="a1" source="p1" target="p2">
    <inscription>
        <graphics><offset x="20" y="0" /></graphics>
        <text>1</text>
    </inscription>
    <graphics><position x="75" y="75" /></graphics>
  </arc>
  </net>
</pnml>
```

Figura 3.11: Definición de PN incorrecta a través de PNML. El arco `a1` sombreado en la figura conecta las plazas `p1` y `p2`.

3.3. Ontologías de redes de Petri

Existen varias propuestas de ontologías de HLPNs. La principal diferencia entre dichas ontologías y la que se presentará en este capítulo es que, al igual que PNML, ninguna de las propuestas permite representar la semántica de ejecución de una red. Es decir, ninguna captura conceptos tan esenciales como las evaluaciones, los modos de transición o los disparos ni tampoco explicitan los axiomas que permiten validar

si la ejecución ha sido correcta.

En [112] Gasevic y Devedzic proponen una ontología de PNs con el objetivo de facilitar el intercambio de redes entre distintas aplicaciones. Esta ontología está basada en el estándar de intercambio de ficheros de PNML [292] y podría considerarse una traducción a OWL [84]. Con ello formalizan el metamodelo de PNML en un lenguaje de lógica descriptiva y, por lo tanto, evitan los problemas anteriormente mencionados que son intrínsecos del formato de intercambio basado en XML. Para el desarrollo de esta ontología los autores han representado el metamodelo de HLPNs en el lenguaje UML y las restricciones entre los conceptos en el lenguaje OCL [10]. Para su paso a OWL utilizaron el editor de ontologías Protégé[1] para modelar PNML y después generar automáticamente el metamodelo en el lenguaje OWL. Aunque la ontología de Gasevic y Devedzic tiene semejanzas con la propuesta en esta tesis doctoral, el enfoque seguido por ambas propuestas es diferente: la ontología de Gasevic y Devedzic está orientada a compartir PNs entre diferentes herramientas transformando la red al formato de la herramienta mediante plantillas XSLT; en cambio nuestra ontología está orientada a una completa especificación de las PNs, incluyendo el concepto de álgebra y la descripción de la semántica operacional que permita capturar la semántica de ejecución de las PNs.

En [236] Songfeng presenta una ontología que describe el modelo estático de una PN a través de dos conceptos de alto nivel y un conjunto de correspondencias entre ellos: el concepto *structure*, que describe los elementos del grafo, y el concepto *algebra*, que representa las anotaciones en los nodos y arcos de la PN. La ontología facilita la reutilización, ya que separa la definición del grafo de su anotación, sin embargo, al no existir firma sintáctica, las correspondencias se deben establecer manualmente. Éste es el principal inconveniente de esta propuesta y reduce considerablemente su posible reutilización.

Otras ontologías han sido propuestas para describir los principales elementos del grafo característico de estas redes y hacer uso de las ventajas del razonamiento del lenguaje de ontología en el que están especificadas. Estas ontologías se utilizan para representar *(i)* la semántica de procesos de negocio para determinar la similaridad entre dos procesos [45] y *(ii)* servicios web semánticos para comprobar la coreografía entre servicios descritos en OWL-S [290].

Del análisis del estado del arte en ontologías de HLPNs se puede concluir que *ninguna* de las ontologías propuestas en la bibliografía . Éste es un gran problema si la ontología se va a utilizar para modelar WFs y dar soporte a su ejecución. Las ontologías anteriormente descritas tampoco cumplen con el estándar ISO/IEC 15909-1, que especifica la estructura y semántica de las HLPNs: ninguna de las ontologías separa la firma sintáctica del álgebra que permite interpretarla y, por lo tanto, posibilitan una menor reutilización de las redes. Finalmente reseñar que la mayoría de las propuestas están orientadas a la transferencia/intercambio de grafos de PNs, de modo que detallan aspectos visuales como posiciones en la pantalla o colores, que tienen más que ver con los editores gráficos que con las propias redes.

[1] http://protege.stanford.edu/

3.4. Ontología de HLPNs

Las HLPNs definen la base a partir de la cual se representan los procesos en esta tesis doctoral: todos los procesos son en último término una HLPN y gracias a ello las herramientas de análisis de las propiedades y del comportamiento de estas redes pueden aplicarse a los WFs. Por lo tanto, un proceso se compone de un conjunto de plazas, transiciones, arcos y anotaciones que representan un grafo de PN. El éxito de estas redes radica precisamente *(i)* en su estructura en forma de grafo, que posibilita modelar sistemas complejos con facilidad, y además *(ii)* en su capacidad de análisis, que permite estudiar tanto la topología de los grafos como su comportamiento.

La ontología que se presenta en este apartado describe las HLPNs como un componente más de conocimiento [266, 276]: especifica de modo explícita y formal la conceptualización de las HLPN, es decir, detalla la semántica de sus conceptos y sus propiedades, de forma que puedan ser entendidos por máquinas a través de un formalismo lógico. El objetivo es que un motor de inferencia pueda razonar sobre las instancias de los conceptos de la ontología, o lo que es lo mismo, que pueda razonar sobre las HLPNs. Además, el modelo que se presenta en este apartado está basado en el estándar ISO/IEC 15909-1 [144] que recoge el conocimiento consensuado de lo que es una HLPN a través de su descripción matemática. Esta ontología captura dicha especificación matemática y la traslada a *(i)* una taxonomía que recoge tanto los elementos estáticos como los dinámicos de estas redes, *(ii)* un conjunto de axiomas que restringen la semántica de los conceptos de dicha taxonomía, y *(iii)* una serie de reglas que modelan el conocimiento sobre la ejecución de la red. Ésta es una de las grandes ventajas de esta propuesta: *con el razonador adecuado es posible definir y ejecutar una PN, ya que la ontología contiene todo el conocimiento necesario para ello.*

Además, esta ontología es compatible con el estándar ISO/IEC 15909-2 [145] para el intercambio de PNs entre distintas herramientas y aplicaciones. Ya que en la actualidad este estándar está siendo implantando en algunas de las más importantes herramientas de PNs, se han creado un conjunto de correspondencias que hacen compatibles los conceptos definidos en la ontología y en PNML.

3.4.1. Modelo estático

Una de las propiedades más interesantes de las PNs es que su definición depende únicamente de componentes estáticos. Estos elementos constituyen las invariantes de las redes, es decir, aquellos elementos que no se ven afectados por la ejecución de la red. Por ejemplo, el grafo de una HLPN o las anotaciones del mismo no se pueden modificar durante una ejecución. Esta propiedad permite independizar la estructura de las redes de los elementos que intervienen en su ejecución y, por lo tanto, reutilizar el mismo grafo en múltiples ejecuciones. La Figura 3.12 representa a través de una red semántica el modelo estático propuesto.

El concepto `HLPN` describe una PN de alto nivel y es el nodo raíz de la taxonomía.

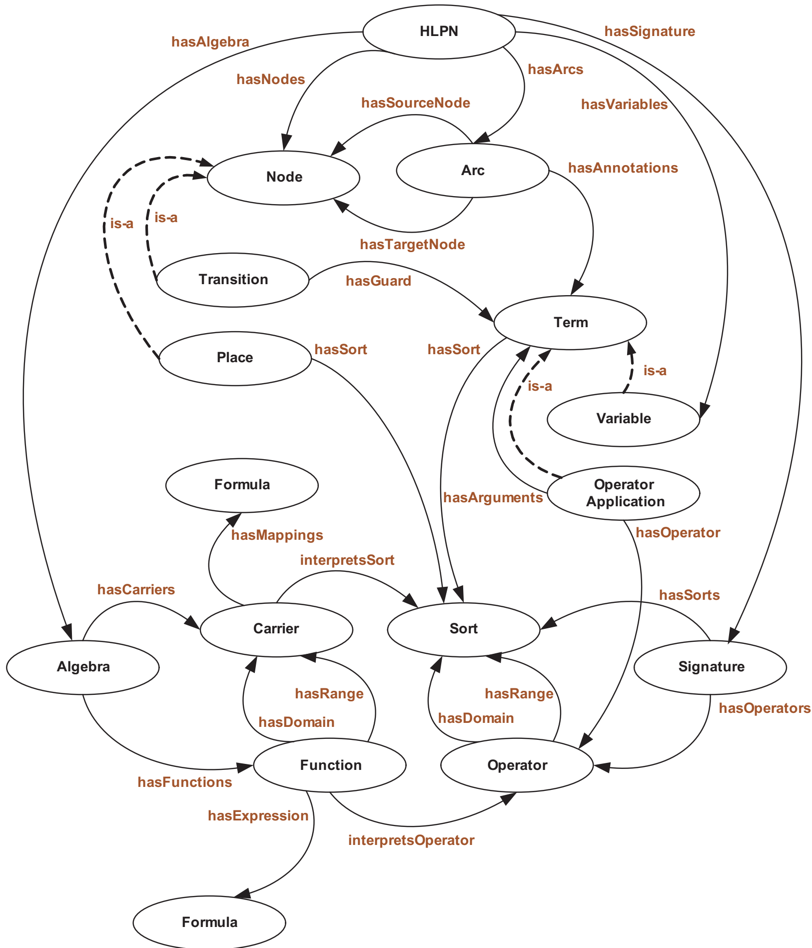


Figura 3.12: Red semántica del modelo estático de la ontología de HLPNs donde las elipses modelan conceptos de la ontología y los arcos las relaciones entre dichos conceptos aunque estos últimos no reflejan la cardinalidad de dichas relaciones. El detalle de esta cardinalidad puede consultarse en el código asociado a cada uno de los conceptos.

Este elemento aglutina al conjunto de conceptos que estructuran una HLPN como un grafo bipartito y dirigido:

```
subClassOf(HLPN, Thing).
```

La definición de una HLPN tiene tres partes: el nombre y la descripción, que aportan información acerca de las características de la red; los nodos y los arcos, que permiten describir su estructura; y finalmente la firma, álgebra y variables, que facilitan su

anotación algebraica. De forma más detallada, los atributos del concepto `HLPN` son los siguientes:

- La propiedad `hasName` se refiere al nombre de la HLPN que puede usarse como identificador de la red:

```
dataProperty(hasName).
domain(hasName, HLPN).
range(hasName, String).
```

- La propiedad `hasDescription` proporciona una breve descripción de la HLPN que resume su funcionalidad y sus particularidades:

```
dataProperty(hasDescription).
domain(hasDescription, HLPN).
range(hasDescription, String).
maxCardinality(hasDescription, 1).
```

- La relación `hasNodes` contiene los vértices o nodos del grafo, que pueden ser plazas o transiciones:

```
objectProperty(hasNodes).
domain(hasNodes, HLPN).
range(hasNodes, Node).
```

- La relación `hasArcs` contiene las aristas del grafo, que se caracterizan por ser dirigidas y conectar nodos de distinto tipo:

```
objectProperty(hasArcs).
domain(hasArcs, HLPN).
range(hasArcs, Arc).
```

- La relación `hasSignature` contiene la firma sintáctica de la red, que agrupa los colores (tipo de las plazas) y operadores que se pueden usar para anotar la HLPN. Es sintáctica, ya que declara los elementos, pero no los dota de ninguna semántica operacional (por ejemplo, los operadores no tienen implementado el código que realiza su funcionalidad):

```
objectProperty(hasSignature).
domain(hasSignature, HLPN).
range(hasSignature, Signature).
cardinality(hasSignature, 1).
```

- La relación `hasAlgebra` aporta la semántica operacional de las anotaciones de la red. Para que una red sea operativa, sus colores y operadores deben tener asociada una semántica, que se explicita en el nivel del lenguaje de representación de la ontología:

```
objectProperty(hasAlgebra).
domain(hasAlgebra, HLPN).
range(hasAlgebra, Algebra).
cardinality(hasAlgebra, 1).
```

- Las anotaciones están descritas mediante términos que pueden ser variables o llamadas a operadores. La relación `hasVariables` contiene las variables con las que se anota la red:

```
objectProperty(hasVariables).
domain(hasVariables, HLPN).
range(hasVariables, Variable).
```

3.4.1.1. Estructura del grafo

Como se comentó anteriormente, las HLPNs son grafos dirigidos con dos tipos de nodos. El concepto `Node` se refiere a los vértices del grafo, independientemente de su tipo, y define una partición exhaustiva y disjunta, de forma que todos los nodos sean o bien una plaza o bien una transición, y que además no puedan ser simultáneamente de ambos tipos:

```
subClassOf(Node, Thing).
subClassOf(Place, Node).
subClassOf(Transition, Node).

oneOf(Node,[Place, Transition]).
disjoint(Place, Transition).
```

Los nodos se identifican por uno o varios nombres. Debido a la complejidad y tamaño de estas redes, su diseño suele realizarse en distintas etapas y típicamente consiste en la unión de distintos modelos. Por ello, cabe la posibilidad de que los nodos estén identificados por distintos nombres:

```
domain(hasName, Node).
```

Para distinguir las marcas asociadas a una plaza, la relación `hasSort` asocia un color al concepto `Place`. El nombre *color* es herencia de las PNs coloreadas, que son el fundamento de las redes de alto nivel:

```
objectProperty(hasSort).
domain(hasSort, Place).
range(hasSort, Sort).
cardinality(hasSort, 1).
```

Por ejemplo, las plazas $p_{1,4}$, $p_{2,5}$ y p_6 de la Figura 3.13 son de tipo *pieza* lo que implica que dichas plazas *únicamente* pueden contener marcas del tipo *pieza*. En cambio, la plaza p_3 contendrá marcas del tipo *maquina*.

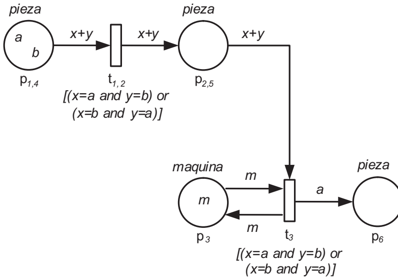


Figura 3.13: PN que representa el estado inicial de una máquina de ensamblaje

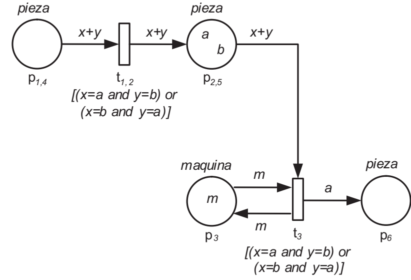


Figura 3.14: Estado de la máquina de ensamblaje tras el disparo de la transición $t_{1,2}$

A diferencia de las PNs de bajo nivel, la ejecución de una transición en una HLPN puede estar condicionada por las precondiciones asociadas a la transición, y por las restricciones de la *regla de disparo de transiciones*. La relación `hasGuard` del concepto `Transition` contendrá la condición de activación asociada a la transición que dará soporte a este comportamiento:

```
objectProperty(hasGuard).
domain(hasGuard, Transition).
range(hasGuard, Term).
cardinality(hasGuard, 1).
```

En PNs, si una transición no está activa, no podrá ocurrir. Por ejemplo, si en la plaza $p_{1,4}$ de la Figura 3.13 no existe al menos una pieza de tipo a y otra de tipo b, la transición $t_{1,2}$ no podrá ocurrir al no verificarse su precondición $(x = a \wedge y = b) \vee (x = b \wedge y = a)$.

El concepto `Arc` representa las aristas del grafo. En PNs, estos arcos son dirigidos: la relación `hasSourceNode` apunta al nodo origen de la arista, mientras que la relación `hasTargetNode` apunta al nodo destino. Al ser redes bipartitas los arcos no pueden establecer una conexión entre nodos del mismo tipo; es decir, un arco no puede conectar dos plazas o dos transiciones:

```
subClassOf(Arc, Thing).

objectProperty(hasSourceNode).
domain(hasSourceNode, Arc).
range(hasSourceNode, Node).
cardinality(hasSourceNode, 1).

objectProperty(hasTargetNode).
domain(hasTargetNode, Arc).
range(hasTargetNode, Node).
cardinality(hasTargetNode, 1).
```

Tabla 3.1: Axiomas que restringen a la estructura del grafo.

Los arcos conectan nodos de distinto tipo.
$\forall A, N_s, N_t\ Arc(A) \wedge hasSourceNode(A, N_s) \wedge hasTargetNode(A, N_t) \rightarrow$ $(Place(N_s) \wedge Transition(N_t)) \vee (Transition(N_s) \wedge Place(N_t))$

La Tabla 3.1 muestra el axioma que impide que un arco relacione nodos del mismo tipo. Es necesario indicar que la Tabla 3.1 y las que se presentarán a lo largo de este capítulo no incluyen toda la axiomática del modelo. Para facilitar su comprensión muchos axiomas se han descrito a través de predicados, ocultando así parte de la complejidad del modelo. Por ejemplo, términos como *subClassOf*, *domain*, *range*, *cardinality*, *maxCardinality* o *minCardinality* son en realidad axiomas del modelo que por simplicidad están descritos a través de predicados. Las reglas encargadas de verificar dichos predicados están descritas en el apéndice C.

Además, los arcos pueden estar anotados. Estas anotaciones juegan un papel fundamental durante la ejecución de estas redes, y su evaluación determinará las marcas a consumir y producir en cada paso de la ejecución. La relación `hasAnnotations` contiene las expresiones que anotan un arco:

```
objectProperty(hasAnnotations).
domain(hasAnnotations, Arc).
range(hasAnnotations, TermMultiset).
cardinality(hasAnnotations, 1).
```

Específicamente, un arco está anotado por un multiconjunto de términos. Al no existir un tipo de dato predefinido para la representación de multiconjuntos, en la presente ontología se han implementado como listas. Por ejemplo, el arco de la Figura 3.13 que conecta las plazas $p_{1,4}$ y $p_{2,5}$ está anotado por el multiconjunto $x + y$. En este caso, la anotación $x + y$ es una abreviatura de $1'x + 1'y$, donde el símbolo $+$ se usa para indicar que el multiconjunto está compuesto por 1 elemento x y 1 elemento y. Por lo tanto, la relación `hasAnnotations` representaría dicha expresión como la lista $[x, y]$. En el caso de que el multiconjunto fuese $2'x + 3'y$, la correspondiente lista pasaría a tener los siguientes elementos $[x, x, y, y, y]$.

3.4.1.2. Firma sintáctica

La firma contiene la sintaxis necesaria para anotar los arcos, las plazas y transiciones de una HLPN. Cada instancia del concepto `Signature` representará una firma y agrupará un conjunto de colores y de operadores a partir de los cuales se construirán los términos que anotarán la red. Se considera sintáctica porque tanto sus colores como operadores lo son: un color es una representación de un tipo de dato y no un tipo de dato real; lo mismo sucede con los operadores, ya que hacen referencia a una funcionalidad, aunque sin llegar a implementarla. Una firma se identifica a través de su nombre y descripción, propiedades `hasName` y `hasDescription` respectivamente:

```
subClassOf(Signature, Thing).

domain(hasName, Signature).
domain(hasDescription, Signature).
```

Las demás propiedades del concepto `Signature` se usan para asociar los colores y los operadores a la firma. Por un lado, la propiedad `hasSorts` se refiere a los colores:

```
objectProperty(hasSorts).
domain(hasSorts, Signature).
range(hasSorts, Sort).
minCardinality(hasSorts, 1).
```

Como se puede apreciar en la representación anterior, la cardinalidad mínima de esta propiedad es uno. Ello se debe a que la firma siempre contendrá al menos el tipo de dato lógico (*boolean*). Esto se debe a que las transiciones están anotadas con precondiciones, por lo que es imprescindible que la firma posea este tipo para poder evaluarlas.

Por otro lado, la relación `hasOperators` se refiere a los operadores de la firma. En este caso, la cardinalidad mínima de esta propiedad es dos, ya que la firma deberá contener siempre a las constantes verdadero y falso:

```
objectProperty(hasOperators).
domain(hasOperators, Signature).
range(hasOperators, Operator).
minCardinality(hasOperators, 2).
```

Por ejemplo, la firma sintáctica de la red representada en la Figura 3.13 podría ser $Sig = (\{boolean, pieza, maquina\}, \{\wedge, \vee, =, a, b, m\})$, donde el conjunto de colores viene dado por $\{boolean, pieza, maquina\}$ y $\{\wedge, \vee, =, a, b, m\}$ es el conjunto de operadores.

Los colores son una representación sintáctica de un tipo de dato y no suelen tener una estructura compleja. Por ello, el concepto `Sort` se define simplemente a partir de su nombre y descripción:

```
subClassOf(Sort, Thing).

domain(hasName, Sort).
domain(hasDescription, Sort).
```

La propiedad `hasName` se refiere al nombre del color y también actúa como identificador que se usará para anotar las plazas de la red. Por lo tanto, si una plaza está anotada con un color solamente podrá almacenar valores de ese color. Por ejemplo, las plazas $p_{1,4}$, $p_{2,5}$ y p_6 de la Figura 3.13 únicamente contendrán marcas con el tipo *pieza*. La relación `hasDescription` proporciona una breve descripción del color.

Un operador se representa a través del concepto `Operator`, el cual se describe a partir de su nombre, rango y dominio:

```
subClassOf(Operator, Thing).

domain(hasName, Operator).
domain(hasDescription, Operator).
```

La propiedad `hasName` almacena el nombre del operador, que lo identifica cuando se emplea para anotar la red. Por ejemplo, el valor a, que hace referencia a una pieza dada en el ejemplo de fabricación de la Figura 3.13, es en realidad un operador. En este caso, este operador es una constante que se usa en la precondición de las transiciones $t_{1,2}$ y t_3. El rango de un operador se expresa por medio de la relación `operatorRange`, que lo asocia con un color:

```
objectProperty(operatorRange).
domain(operatorRange, Operator).
range(operatorRange, Sort).
cardinality(operatorRange, 1).
```

Por ejemplo, la constante a del ejemplo de fabricación tiene un rango de tipo *pieza*. La relación `operatorDomain` se refiere al dominio de un operador y se expresa como una lista de colores, donde cada uno de sus elementos representa uno de los tipos del producto cartesiano:

```
objectProperty(operatorDomain).
domain(operatorDomain, Operator).
range(operatorDomain, SortList).
cardinality(operatorDomain, 1).
```

Por ejemplo, el operador $\wedge$ empleado en la precondición de las transiciones $t_{1,2}$ y t_3 puede definirse como $\wedge$: *boolean* $\times$ *boolean* $\rightarrow$ *boolean*. En lógica de predicados esta definición quedaría como $operatorDomain(\wedge, [boolean, boolean])$. Para las constantes, como es el caso del operador a, el dominio es el conjunto vacío y en la ontología se representará como una lista vacía: $operatorDomain(a, [])$.

La Tabla 3.2 recoge los axiomas que restringen la firma sintáctica de una HLPN. Estos axiomas permiten chequear que la firma es una firma lógica (*boolean signature*, en inglés) y que tanto las plazas como los operadores se construyen a partir de colores de la firma.

3.4.1.3. Anotaciones

El concepto `Term` proporciona una superclase para la descripción de los términos que se utilizan para anotar los arcos y las transiciones de las HLPNs:

```
subClassOf(Term, Thing).

domain(hasSort, Term).
```

Tabla 3.2: Axiomas asociados a la firma sintáctica.

La firma sintáctica de la red incluye el tipo de dato *boolean*.
$\forall H, S\ HLPN(H) \wedge Signature(S) \wedge hasSignature(H, S) \rightarrow Sort(boolean) \wedge hasSorts(S, boolean)$
La constante *true* está definida dentro de los operadores de la red.
$\forall H, S\ \ Signature(S) \wedge hasSorts(S, boolean) \rightarrow Operator(true) \wedge operatorDomain(true, []) \wedge operatorRange(true, boolean) \wedge hasOperators(S, true)]$
La constante *false* está definida dentro de los operadores de la red.
$\forall H, S\ \ Signature(S) \wedge hasSorts(S, boolean) \rightarrow Operator(false) \wedge operatorDomain(false, []) \wedge operatorRange(false, boolean) \wedge hasOperators(S, false)$
Los tipos de datos de las plazas pertenecen a la firma sintáctica de la red.
$\forall S, S_o, H, P\ Signature(S) \wedge Sort(S_o) \wedge HLPN(H) \wedge Place(P) \wedge hasSignature(H, S) \wedge hasNodes(H, P) \wedge hasSort(P, S_o) \rightarrow hasSorts(S, S_o)$
El dominio de un operador es una lista que contiene tipos de datos que pertenecen a su misma firma sintáctica.
$\forall O, S, S_o, D\ Operator(O) \wedge Sort(S_o) \wedge Signature(S) \wedge hasOperators(S, O) \wedge list(D, Sort) \wedge operatorDomain(O, D) \wedge member(S_o, D) \rightarrow hasSorts(S, S_o)$
El rango de un operador es de un tipo de dato que pertenece a su misma firma sintáctica.
$\forall O, S, S_o\ Operator(O) \wedge Sort(S_o) \wedge Signature(S) \wedge hasOperators(S, O) \wedge operatorRange(O, S_o) \rightarrow hasSorts(S, S_o)$

La relación `hasSort` se refiere al color asociado a la expresión y se usa para verificar que el término está tipado correctamente. Por ejemplo, cuando un término representa la condición de activación de una transición su tipo debe ser obligatoriamente lógico (*boolean*). Además, el color del término debe pertenecer a la firma sintáctica, ya que en caso contrario el álgebra no podría interpretar la anotación y, por lo tanto, la red sería inconsistente.

Los términos son una parte fundamental del dinamismo de estas redes: se usan para restringir la activación de las transiciones, establecer los modos de transición y calcular el resultado del disparo de una transición. Los términos representan expresiones que pueden contener variables (por ejemplo, x, y) y llamadas a operadores (por ejemplo $f(x, y)$). Para modelar estas expresiones concretas el concepto `Term` se especifica en dos sublcases que definen una jerarquía exhaustiva y disjunta:

```
subClassOf(Variable, Term).
subClassOf(OperatorApplication,Term).

oneOf(Term,[Variable,OperatorApplication]).
disjoint(Variable, OperatorApplication).
```

El concepto `Variable` representa una variable de programación cuyo nombre se especifica por medio de la propiedad `hasName` y que actúa como identificador de la variable en la anotación de la red. Por ejemplo, las dos variables usadas en la Figura 3.13 se representan mediante x e y.

```
domain(hasName, Variable).
```

El concepto `OperatorApplication` representa la llamada a un operador y se usa para componer términos complejos. La relación `hasOperator` se refiere al operador que se debe invocar, mientras que la relación `hasArguments` hace referencia a la lista de argumentos de dicho operador:

```
objectProperty(hasOperator).
domain(hasOperator, OperatorApplication).
range(hasOperator, Operator).
cardinality(hasOperator, 1).

objectProperty(hasArguments).
domain(hasArguments, OperatorApplication).
range(hasArguments, TermList).
cardinality(hasArguments, 1).
```

Es importante destacar que el color del término debe coincidir con el rango del operador. Igualmente, el color de cada uno de los términos que configuran la lista de argumentos deben coincidir con el correspondiente color en el dominio del operador. Por ejemplo, el término $(x = a \wedge y = b) \vee (x = b \wedge y = a)$ existe una llamada al operador $\vee$ que tiene asociados dos argumentos: los términos $(x = a \wedge y = b)$ y $(x = b \wedge y = a)$. Es fácil de comprobar que ambos términos son también de tipo lógico, lo cual coincide con el tipo del operador $\vee : boolean \times boolean \rightarrow boolean$.

La Tabla 3.3 recoge formalmente los axiomas que se han ido introduciendo en este apartado y que restringen la anotación de los arcos y de las transiciones de la red.

3.4.1.4. Álgebra

La interpretación de una firma sintáctica, y consecuentemente de las anotaciones de la red, está a cargo del álgebra. La propiedad `hasName` se refiere al nombre mediante el cual se identificará el álgebra, mientras que `hasDescription` contendrá un breve resumen del contenido e información relevante de este concepto.

```
subClassOf(Algebra, Thing).

domain(hasName, Algebra).
domain(hasDescription, Algebra).
```

La relación `hasCarriers` se refiere al conjunto de tipos de datos del álgebra, cada uno de los cuales se denomina *soporte* (del inglés *carrier*). Al contrario que los colores, los soportes tendrán asociadas instancias/individuos. Por ejemplo, un soporte que implemente el concepto asociado a los usuarios de un sistema tendrá una instancia por cada usuario registrado en el sistema.

```
objectProperty(hasCarriers).
```

Tabla 3.3: Axiomas que restringen a la anotación del grafo

Las precondiciones de las transiciones son de tipo *boolean*.
$\forall T, G\ Transition(T) \wedge Term(G) \wedge hasGuard(T,G) \rightarrow hasSort(G, boolean)$
Los anotaciones de los arcos de salida de una plaza tienen el tipo de esa plaza.
$\forall A, P, S_P, S_T, T, L\ Arc(A) \wedge Place(P) \wedge Sort(S_p) \wedge Sort(S_T) \wedge Term(T) \wedge hasSort(P, S_P) \wedge hasSort(T, S_T) \wedge hasSourceNode(A, P) \wedge hasAnnotations(A, L) \wedge member(T, L) \rightarrow S_T = S_P$
Los anotaciones de los arcos de entrada de una plaza tienen el tipo de esa plaza.
$\forall A, P, S_P, S_T, T, L\ Arc(A) \wedge Place(P) \wedge Sort(S_p) \wedge Sort(S_T) \wedge Term(T) \wedge hasSort(P, S_P) \wedge hasSort(T, S_T) \wedge hasTargetNode(A, P) \wedge hasAnnotations(A, L) \wedge member(T, L) \rightarrow S_T = S_P$
El término que representa una llamada a un operador tiene el mismo tipo que dicho operador.
$\forall O, S, T\ OperatorApplication(T) \wedge Operator(O) \wedge Sort(S) \wedge \wedge hasOperator(T, O) \wedge hasSort(O, S) \rightarrow operatorRange(O, S)$
La lista de argumentos de una llamada a un operador contiene términos con el mismo tipo que el dominio del operador.
$\forall L1, L2, O, T\ Operator(O) \wedge OperatorApplication(T) \wedge hasOperator(T, O) \wedge \wedge argumentsSort(T, L1) \wedge domain(O, L2) \rightarrow L1 = L2$

```
domain(hasCarriers, Algebra).
range(hasCarriers, Carrier).
minCardinality(hasCarriers, 1).
```

La cardinalidad mínima de la relación `hasCarriers` es uno, ya que el álgebra siempre contendrá la implementación (es decir, el soporte) del tipo sintáctico `boolean`.

La relación `hasFunctions` apunta a las funciones implementadas en el álgebra, que como mínimo serán dos, ya que siempre existirán las implementaciones de las constantes *true* y *false*:

```
objectProperty(hasFunctions).
domain(hasFunctions, Algebra).
range(hasFunctions, Function).
minCardinality(hasFunctions, 2).
```

Tanto los soportes como las funciones se identifican a través de la propiedad `hasName` y contienen información adicional acerca de sus características más relevantes a través de la propiedad `hasDescription`:

```
subClassOf(Carrier, Thing).
subClassOf(Function, Thing).

domain(hasName, Carrier).
domain(hasDescription, Carrier).
domain(hasName, Function).
domain(hasDescription, Function).
```

En este contexto, un álgebra proporciona una semántica operacional a cada uno de los colores y de los operadores de la firma, es decir, asocia soportes a colores y funciones a operadores a través de las relaciones `interpretsSort` e `interpretsOperator`, respectivamente:

```
objectProperty(interpretsSort).
domain(interpretsSort, Carrier).
range(interpretsSort, Sort).
maxCardinality(interpretsSort, 1).

objectProperty(interpretsOperator).
domain(interpretsOperator, Function).
range(interpretsOperator, Operator).
maxCardinality(interpretsOperator, 1).
```

Cuando una función interpreta a un operador es necesario que el dominio y rango de la función coincidan con los del operador. Así, la relación `functionDomain` contiene una lista de soportes donde cada soporte interpreta el correspondiente color dentro del dominio del operador. Lo mismo sucederá con la relación `functionRange` para el rango de la función y del operador. Por ejemplo, si la función $autenticar : cadena \times cadena \to logico$ interpreta al operador $login : string \times string \to boolean$ es necesario que los tipos *cadena* y *logico* sean los soportes de los colores *string* y *boolean*, respectivamente.

```
objectProperty(functionDomain).
domain(functionDomain, Function).
range(functionDomain, CarrierList).
cardinality(functionDomain, 1).

objectProperty(functionRange).
domain(functionRange, Function).
range(functionRange, Carrier).
cardinality(functionRange, 1).
```

La separación entre *(i)* soportes y colores y entre *(ii)* funciones y operadores permite utilizar distintas álgebras (nivel semántico) para una misma firma (nivel sintáctico). Esto también facilita la definición de HLPNs independientemente del álgebra usada en su ejecución. Por ejemplo, la red de la Figura 3.13 podría usarse para controlar otros procesos distintos al proceso de ensamblaje para el cual ha sido diseñada. Por ejemplo, si esta red se quiere usar para mezclar dos componentes de un pienso para animales, bastaría con usar un álgebra donde el soporte *compuesto* interprete el color *pieza*, el soporte *mezclador* interprete el color *máquina*, y un par de instancias del soporte *compuesto* interpretasen las constantes a y b.

La expresividad de las funciones del álgebra estará limitada por la expresividad del modelo de representación del lenguaje de la ontología en la que se implementa. Por ejemplo, las propiedades `hasMappings` y `hasExpression` dependerán del lenguaje en que se implementen, en la medida en que si se describen con F-Logic serán mucho más expresivas que en OWL (que tiene un conjunto predefinido y limitado de funciones).

```
objectProperty(hasMappings).
domain(hasMappings, Carrier).
range(hasMappings, Formula).
cardinality(hasMappings, 1).
```

La propiedad `hasMappings` se utiliza para renombrar o definir correspondencias entre soportes y colores y entre funciones y operadores. Por ejemplo, se puede usar para establecer correspondencias entre formatos distintos de fechas.

```
objectProperty(hasExpression).
domain(hasExpression, Function).
range(hasExpression, Formula).
cardinality(hasExpression, 1).
```

La propiedad `hasExpression` permite definir una fórmula que actuará como descripción operacional de la función (o lo que es lo mismo, será su implementación). Cuando el motor de HLPNs evalúa un término que hace uso de un operador, accede a la función que interpreta dicho operador y ejecuta su expresión con los datos correspondientes.

La Tabla 3.4 recoge los axiomas que aseguran que todos los colores y operadores de la firma sintáctica son interpretados por los soportes y las funciones incluidas en el álgebra.

3.4.2. Modelo dinámico

El modelo presentado en el apartado anterior describe las HLPNs independientemente de su estado. Es una propiedad deseable en cualquier modelo y que además permite razonar acerca de la estructura y definiciones de la red sin tener en cuenta o depender de su estado. Como complemento a este modelo estático en este apartado se describen dos nuevos modelos que permiten capturar el *estado* y la *ejecución* de las HLPNs. A través de ellos se podrá saber tanto el estado de la red en un determinado instante de tiempo como la sucesión de eventos que la han llevado a dicho estado. Además, estos modelos se han diseñado con la intención de facilitar la depuración de la ejecución. Para cada evento se guardará el valor que toman cada una de las variables y cada uno de los términos que anotan los arcos relacionados con las transiciones ejecutadas.

3.4.2.1. Estado

El modelo de situación se representa en la Figura 3.15 y describe aquellos componentes que definen el estado de una HLPN, el cualestá compuesto por las marcas localizadas en cada una de las plazas. Este modelo, por tanto, describe los conceptos que permiten relacionar una plaza con sus marcas en cada instante de ejecución de la red. El concepto `PlaceMarking` define el estado de una plaza:

```
subClassOf(PlaceMarking, Thing).
```

Tabla 3.4: Axiomas que restringen al álgebra de una HLPN

En una HLPN, cada tipo de datos de la firma sintáctica (Sort) tiene un tipo de datos asociado dentro del álgebra (Carrier).
$\forall A, H, S, S_o \; Algebra(A) \wedge HLPN(H) \wedge Signature(S) \wedge Sort(S_o) \wedge hasSignature(H, S) \wedge hasAlgebra(H, A) \wedge hasSorts(S, S_o) \rightarrow \exists C \; Carrier(C) \wedge hasCarriers(A, C) \wedge interpretsSort(C, S_o)$
En una HLPN, cada operador de la firma sintáctica tiene una función asociada dentro del álgebra.
$\forall A, H, S, O Algebra(A) \wedge HLPN(H) \wedge Signature(S) \wedge Operator(O) \wedge hasSignature(H, S) \wedge hasAlgebra(H, A) \wedge hasOperators(S, O) \rightarrow \exists F \; Function(F) \wedge hasFunctions(A, F) \wedge interpretsOperator(F, O)$
El dominio de una función está compuesto por una lista de tipos de datos (Carrier) del álgebra. Cada uno de estos tipos tiene asociado dentro del contexto de la firma sintáctica a su par correspondiente, de forma que el dominio de la función y del operador asociado a la función coincidan.
$\forall F, L_F, L_O, O, S \; Function(F) \wedge Operator(O) \wedge Sort(S) \wedge interpretsOperator(F, O) \wedge functionDomain(F, L_F) \wedge operatorDomain(O, L_O) \wedge member(S, L_O) \rightarrow \exists C \; Carrier(C) \wedge member(C, L_F) \wedge interpretsSort(C, S)$
El rango de una función está definido por un tipo de dato (Carrier) del álgebra que interpreta a su par correspondiente dentro del contexto de la firma sintáctica, de forma que el rango de la función y del operador asociado a la función coincidan.
$\forall F, O, S \; Function(F) \wedge Operator(O) \wedge Sort(S) \wedge interpretsOperator(F, O) \wedge functionRange(F, S) \rightarrow \exists C \; Carrier(C) \wedge interpretsSort(C, S)$

```
objectProperty(hasPlace).
domain(hasPlace, PlaceMarking).
range(hasPlace, Place).
cardinality(hasPlace, 1).

objectProperty(hasTokens).
domain(hasTokens, PlaceMarking).
range(hasTokens, FunctionMultiset).
maxCardinality(hasTokens, 1).
```

La relación `hasPlace` se refiere a la plaza cuyo estado está siendo representado, mientras que la relación `hasTokens` contiene una lista de valores. De esta forma, el concepto `PlaceMarking` asocia una plaza al conjunto de marcas que contiene. Como ya se comentó a lo largo de este capítulo, los valores (o constantes) se representan como funciones sin dominio, por lo que la relación `hasTokens` apunta a una lista (multiconjunto) de funciones. Por ejemplo, suponiendo que la red de la Figura 3.13 se utiliza para mezclar piensos, el valor a se definiría como una función $a : [] - > compuesto$, es decir, sería una constante (no tiene dominio) con rango de tipo *compuesto*. Los valores almacenados en una plaza han de cumplir una restricción adicional: su rango ha de ser *compatible* con el color de la función, o lo que es lo mismo, el rango soporte de la función debe interpretar el color de la plaza. En el ejemplo anterior esta restricción se cumple, ya que el soporte *compuesto* interpreta el color *pieza*.

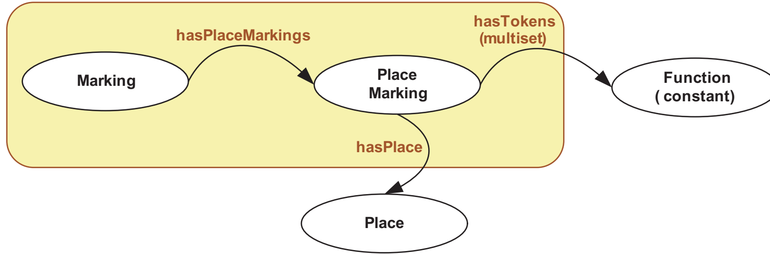


Figura 3.15: Red semántica del modelo de situación de las HLPNs

Tabla 3.5: Axiomas que restringen al estado de la red

El estado de la red contiene un único estado asociado a cada plaza.
$\forall E, H, M, P, M_1, M_2\ Marking(M) \wedge Place(P) \wedge PlaceMarking(M_1) \wedge PlaceMarking(M_2) \wedge hasPlaceMarkings(M, M_1) \wedge hasPlaceMarkings(M, M_2) \wedge hasPlace(M_1, P) \wedge hasPlace(M_2, P) \rightarrow M_1 = M_2$

Los valores asociados a una plaza son “compatibles” con el color de dicha plaza.
$\forall C, F, M, P, L, S\ Carrier(C) \wedge Function(F) \wedge PlaceMarking(M) \wedge Place(P) \wedge hasPlace(M, P) \wedge hasTokens(M, L) \wedge member(F, L) \wedge functionRange(F, C) \wedge hasSort(P, S) \rightarrow interpretsSort(C, S)$

Las marcas de las plazas son constantes.
$\forall F, M, L\ Function(F) \wedge PlaceMarking(M) \wedge hasTokens(M, L) \wedge member(F, L) \rightarrow functionDomain(F, [])$

El estado completo de la red se representa por medio del concepto `Marking`, que agrupa el estado de cada una de las plazas a través de la relación `hasPlaceMarkings`:

```
subClassOf(Marking, Thing).

objectProperty(hasPlaceMarkings).
domain(hasPlaceMarkings, Marking).
range(hasPlaceMarkings, PlaceMarking).

inverseOf(hasMarking, hasPlaceMarkings).
```

Por ejemplo, el estado inicial de la red representada en la Figura 3.13 contendría las instancias del concepto `PlaceMarking` asociados a las plazas $p_{1,4}$ y p_3. Es necesario comentar que el estado de cada plaza (*place marking*) se incluye una *única* vez en el estado correcto de la red (*marking*). En caso contrario, una plaza podría estar en varios estados en el mismo instante de tiempo, lo cual sería inconsistente con la especificación ISO/IEC 15909-1.

La Tabla 3.5 recoge los axiomas que aseguran que tanto el concepto `Marking` como el concepto *PlaceMarking* capturan la semántica de estado previamente detallada.

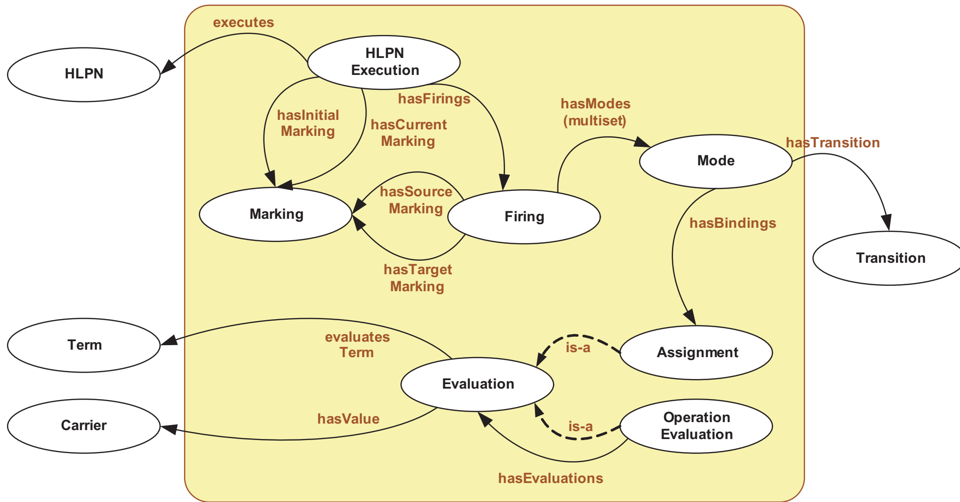


Figura 3.16: Red semántica del modelo de ejecución de las HLPNs

3.4.2.2. Ejecución

El modelo de ejecución captura el flujo de información definido entre dos estados de la red. Específicamente, detalla el camino de eventos a seguir para alcanzar un estado dado, entendiendo por evento al disparo u ocurrencia de una transición. La Figura 3.16 representa este modelo, cuya raíz es el concepto `HLPNExecution` que asocia una determinada instancia de ejecución a una única red a través de la relación `executes`:

```
subClassOf(HLPNExecution, Thing).

objectProperty(executes).
domain(executes, HLPNExecution).
range(executes, HLPN).
cardinality(executes, 1).
```

Cada ejecución parte de un estado inicial (relación `hasInitialMarking`) y se encuentra en un estado actual (relación `hasCurrentMarking`):

```
objectProperty(hasInitialMarking).
domain(hasInitialMarking, HLPNExecution).
range(hasInitialMarking, Marking).
cardinality(hasInitialMarking, 1).

objectProperty(hasCurrentMarking).
domain(hasCurrentMarking, HLPNExecution).
range(hasCurrentMarking, Marking).
cardinality(hasCurrentMarking, 1).
```

Además, el concepto `HLPNExecution` también contiene el conjunto de eventos que han producido un cambio de estado a través de la relación `hasFirings`. Esta relación almacena ordenadamente cada disparo de una transición de una lista ordenada, y con ello se puede seguir la traza desde el estado inicial hasta el estado actual de ejecución.

```
objectProperty(hasFirings).
domain(hasFirings, HLPNExecution).
range(hasFirings, FiringList).
maxCardinality(hasFirings, 1).
```

El concepto `Firing` (disparo en castellano) es el núcleo del modelo de ejecución, dado que captura el cambio de estados producido por el disparo de un conjunto de transiciones:

```
subClassOf(Firing, Thing).
```

Un disparo se refiere a un paso de ejecución entre un estado inicial y un estado final. Las relaciones `hasSourceMarking` y `hasTargetMarking` apuntan respectivamente a estos estados:

```
objectProperty(hasSourceMarking).
domain(hasSourceMarking, Firing).
range(hasSourceMarking, Marking).
cardinality(hasSourceMarking, 1).

objectProperty(hasTargetMarking).
domain(hasTargetMarking, Firing).
range(hasTargetMarking, Marking).
cardinality(hasTargetMarking, 1).
```

Por ejemplo, supóngase que el disparo f_1 representa el cambio de estado entre las figuras 3.13 y 3.14. En este caso, f_1 detalla el paso de un estado $m_1 : [p_{1,4} = \{a, b\}, p_{2,5} = \emptyset, p_3 = \{m\}, p_6 = \emptyset]$ a un estado $m_2 : [p_{1,4} = \emptyset, p_{2,5} = \{a, b\}, p_3 = \{m\}, p_6 = \emptyset]$.

Es necesario precisar que el concepto `Firing` *no* está restringido a la ejecución de una *única* transición, sino que puede referirse a un multiconjunto de transiciones; es decir, el disparo se identifica con un paso de ejecución, independientemente del número de transiciones ejecutadas en él. Es más, en un mismo paso, una misma transición puede ejecutarse varias veces, por lo que es necesario distinguir cada una de esas ejecuciones. Por ello, la relación `hasModes` apunta a una lista con cada uno de los modos de transición ejecutados:

```
objectProperty(hasModes).
domain(hasModes, Firing).
range(hasModes, ModeMultiset).
cardinality(hasModes, 1).
```

Un modo de transición (concepto `Mode`) asocia una transición a un conjunto de asignaciones de variables. Específicamente, un modo indica que una transición está *activa* (se cumple su precondición) cuando un conjunto de variables toman un determinado valor. La relación `hasTransition` se refiere a la transición activa, mientras que la relación `hasBindings` apunta al conjunto de asignaciones que permiten activar dicha transición:

```
subClassOf(Firing, Thing).

objectProperty(hasTransiton).
domain(hasTransiton, Mode).
range(hasTransiton, Transition).
cardinality(hasTransiton, 1).

objectProperty(hasBindings).
domain(hasBindings, Mode).
range(hasBindings, Assignment).
cardinality(hasBindings, 1).
```

En el ejemplo de ensamblaje de la Figura 3.13 un posible modo para la transición $t_{1,2}$ tendría los siguientes valores $x = a$ e $y = b$. Este mismo modo podría usarse en el diaparo f_1 que establece el paso de la red de un estado m_1 a un estado m_2. Dado que un disparo permite la ejecución de varios modos de transición, es necesario asegurar que:

- Una variable no puede tomar valores diferentes en un mismo disparo. Por lo tanto, los distintos modos de transición del paso compartirán las mismas asignaciones.
- Existen suficientes valores en las plazas de entrada para satisfacer la ejecución concurrente de los modos que componen el paso. Por ejemplo, si el estado inicial de la red de la Figura 3.13 fuese $m_1 : [p_{1,4} = \{a, a, b, b\}, p_{2,5} = \emptyset, p_3 = \{m\}, p_6 = \emptyset]$, en un paso se podría disparar dos veces la transición $t_{1,2}$ con los valores $x = a$ e $y = b$.

El cambio entre dos estados está definido por los valores que se han consumido y aquellos que se han producido después de la ejecución de un evento. Para completar el modelo de ejecución, la ontología también captura la *evaluación* de los términos que indican qué marcas es necesario consumir y cuáles producir. De esta forma, es posible comprobar la corrección del cambio de estado verificando que, para los modos de transición ejecutados, las marcas eliminadas se corresponden con las evaluaciones de los arcos de entrada y que las nuevas marcas se corresponden con las evaluaciones de los arcos de salida de los modos de transición ejecutados. Cuando se calcula el valor de un término, esta ontología permite guardar su evaluación a través del concepto `Evaluation`:

```
subClassOf(Evaluation, Thing).
```

```
objectProperty(evaluatesTerm).
domain(evaluatesTerm, Evaluation).
range(evaluatesTerm, Term).
cardinality(evaluatesTerm, 1).

objectProperty(hasValue).
domain(hasValue, Evaluation).
range(hasValue, Carrier).
cardinality(hasValue, 1).
```

La relación `evaluatesTerm` se refiere al término evaluado mientras que la relación `hasValue` guarda su valor. Además, el modelo contempla dos tipos de evaluaciones en función de si el término es una variable o una llamada a un operador:

```
subClassOf(Assignment, Evaluation).
subClassOf(OperatorEvaluation, Evaluation).

allValuesFrom(evaluatesTerm, Assignment, Variable).
allValuesFrom(evaluatesTerm, OperatorEvaluation, OperatorApplication).

oneOf(Evaluation,[Assignment,OperatorEvaluation]).
disjoint(Assignment, OperatorEvaluation).
```

El concepto `Assignment` captura la evaluación de una variable, mientras que el concepto `OperatorEvaluation` hace lo propio con la evaluación de la llamada a un operador, la cual extiende el concepto `Evaluation` con la relación `hasEvaluations`, que se refiere al valor que toma cada uno de los parámetros de la llamada al operador. Como cada uno de esos parámetros es un término, el valor se determinará a partir de su evaluación:

```
objectProperty(hasEvaluations).
domain(hasEvaluations, OperatorEvaluation).
range(hasEvaluations, EvaluationList).
maxCardinality(hasEvaluations, 1).
```

La Tabla 3.6 recoge los axiomas que restringen la semántica del modelo de ejecución representado en la Figura 3.16.

3.4.3. Reglas

El desarrollo de esta ontología se ha fundamentado en una aproximación homogénea donde la taxonomía, los axiomas y las reglas están representadas en un mismo lenguaje lógico. Específicamente, se usó el lenguaje F-Logic [152]. Este lenguaje está basado en el formalismo de la lógica de marcos y permite combinar reglas y taxonomías de forma natural. Así, se han definido reglas que permiten *(i)* gestionar la creación, modificación y borrado de los elementos que componen estas redes; y *(ii)* ejecutarlas. En este sentido, la capa de reglas está unida al formalismo de lógica de marcos y no es

Tabla 3.6: Axiomas que restringen a la ejecución de la red

El resultado de una evaluación tiene un tipo que *interpreta* el color del término que evalúa.
$\forall E,T,S\ Evaluation(E) \wedge Term(T) \wedge Sort(S) \wedge evaluatesTerm(E,T) \wedge hasValue(E,V) \wedge hasSort(T,S) \rightarrow \exists C\ instance(V,C) \wedge Carrier(C) \wedge interpretsSort(C,S)$

En un modo de transición, la evaluación de la precondición de la transición tiene el valor *true* para el conjunto de asignaciones definido.
$\forall E,T,P,M,B\ Evaluation(E) \wedge Transition(T) \wedge Term(P) \wedge Mode(M) \wedge evaluatesTerm(E,P) \wedge hasTransition(M,T) \wedge hasGuard(T,P) \wedge hasValue(E,V) \wedge hasBindings(M,B) \wedge validBinding(B) \rightarrow V = true$

directamente aplicable a razonadores basados en lenguajes con una menor expresividad (como, por ejemplo, OWL). Sin embargo, la adaptación a las características de otro razonador es posible, ya que las reglas expresadas en F-Logic se pueden trasladar a lenguajes lógicos más conocidos que comparten una sintaxis similar, como Prolog. Por esta razón, y para facilitar su comprensión, las reglas se expresarán en Prolog.

3.4.3.1. Reglas para la evaluación de términos

El dinamismo de las HLPNs está determinado por la evaluación de sus anotaciones. Las reglas que evalúan las llamadas a los operadores y a las variables son los elementos más representativos de esta categoría.

Por una parte, las asignaciones de las variables que anotan los arcos de las redes se infieren con la siguiente regla:

```
Assignment(asi(?_var, ?_tok)) :-
  hasCurrentMarking(?_exe, ?_mar),
  hasMarking(?_pma, ?_mar),
  hasPlace(?_pma, ?_pla),
  hasTokens(?_pma, ?_tol),
  hasSourceNode(?_arc, ?_pla),
  hasAnnotation(?_arc, ?_tel),
  useTerm(?_arc, ?_var),
  hasSort(?_pla, ?_sor),
  hasSort(?_var, ?_sor),
  member(?_tok, ?_tol).
```

Esta regla requiere que los arcos tengan como nodo de inicio una plaza que contenga una marca. Por lo tanto, se limita la creación de asignaciones a las marcas existentes en la red, para así no sobrecargar el razonador con asignaciones y evaluaciones innecesarias. El predicado `useTerm` se utiliza para comprobar si el término que anota el arco o alguno de sus (sub)términos contienen la variable a evaluar.

Cada vez que la regla anterior se evalúa con éxito, se genera un nuevo hecho (en este caso una asignación) con el identificador `asi(?_var, ?_tok)`, donde `?_var`

y `?_tok` indican la variable y el valor que conforman la asignación respectivamente. Esta asignación se completa asociando al identificador las propiedades `evaluatesTerm` y `hasValue` a través de las siguientes reglas:

```
evaluatesTerm(asi(?_var, ?_tok), ?_var) :-
  Assignment(asi(?_var, ?_tok)).

hasValue(asi(?_var, ?_tok), ?_tok) :-
  Assignment(asi(?_var, ?_tok)).
```

Por otra parte, las llamadas a operadores se evalúan con dos reglas. La primera contempla la evaluación de las constantes (la propiedad `hasArguments` es una lista vacía), y en ella la llamada a la constante está representada por la expresión `?_nam([], ?_tok)`, donde la variable `?_nam` indica el nombre de la función a invocar, el símbolo `[]` indica que la llamada no tiene parámetros, y la variable `?_tok` contiene el resultado de la evaluación:

```
OperatorEvaluation(ope(?_opa, ?_tok, [])) :-
  hasOperator(?_opa, ?_opr),
  hasArguments(?_opa, []),
  interpretsOperator(?_fun, ?_opr),
  hasName(?_fun, ?_nam),
  ?_nam([], ?_tok).
```

La segunda regla se refiere a la evaluación de los operadores n-arios con un número de argumentos mayor que cero:

```
OperatorEvaluation(ope(?_opa, ?_tok, ?_evl)) :-
  HLPNExecution(?_exe),
  hasHLPN(?_exe, ?_net),
  hasCurrentMarking(?_exe, ?_mar),
  hasNodes(?_net, ?_tra),
  Transition(?_tra),
  hasInputTokens(?_tra, ?_mar),
  useTerm(?_tra, ?_opa),
  hasOperator(?_opa, ?_opr),
  hasArguments(?_opa, ?_arl),
  getArgumentListEvaluation(?_arl, ?_evl),
  formatOperatorArguments(?_evl, ?_cal),
  interpretsOperator(?_fun, ?_opr),
  hasName(?_fun, ?_nam),
  ?_nam(?_cal, ?_tok).
```

En esta segunda regla se filtran los operadores a evaluar tomando únicamente aquellos relacionados con una transición que contiene marcas en cada una de sus plazas de entrada. Por un lado, el predicado `hasInputTokens` indica las transiciones `?_tra` activas en un determinado estado `?_mar`. Por el otro, el predicado `useTerm` indica los términos que se relacionan con cada transición, es decir, los términos que anotan sus arcos de entrada y de salida, y sus precondiciones. En el caso de que el término

coincida con una llamada a un operador se inferirá una evaluación. Para ello, el predicado `getArgListEvaluation` asocia la lista de argumentos `?_arl` con una lista de sus posibles evaluaciones, y el predicado `formatOperatorArguments` relaciona estas evaluaciones con una lista con los valores de la llamada al operador. Finalmente, esta llamada se realiza a través de la expresión `?_nam(?_cal, ?_tok)`. Al igual que en el caso de las asignaciones, el identificador generado por las dos reglas anteriores se usará para completar las propiedades asociadas a las evaluaciones:

```
evaluatesTerm(ope(?_opa, ?_tok, ?_evl), ?_opa) :-
  OperatorEvaluation(ope(?_opa, ?_tok, ?_evl)).

hasValue(ope(?_opa, ?_tok, ?_evl), ?_tok) :-
  OperatorEvaluation(ope(?_opa, ?_tok, ?_evl)).

hasEvaluations(ope(?_opa, ?_tok, ?_evl), ?_evl) :-
  OperatorEvaluation(ope(?_opa, ?_tok, ?_evl)).
```

3.4.3.2. Reglas para el cálculo de los modos de transición

La siguiente regla se encarga de inferir los modos *aplicables* a cada transición de la red:

```
Mode(tmo(?_tra, ?_asl)) :-
  hasGuard(?_tra, ?_gua),
  useVariableList(?_tra, ?_val),
  evaluatesTerm(?_eva, ?_gua),
  hasValue(?_eva, true),
  hasAssignmentList(?_eva, ?_evl),
  evaluatesVariableList(?_eva, ?_vrl),
  getModeAssignments(?_val, ?_vrl, ?_evl, ?_asl).

hasTransition(tmo(?_tra, ?_asl), ?_tra) :-
  Mode(tmo(?_tra, ?_asl)).

hasAssignments(tmo(?_tra, ?_asl), ?_asl) :-
  Mode(tmo(?_tra, ?_asl)).
```

Esta regla obtiene las transiciones que verifican sus precondiciones y las asignaciones que son resultado de la evaluación de dichas condiciones. Finalmente, el predicado `getModeAssignments` completa la lista de asignaciones del modo para aquellas variables que no están en las precondiciones. Aunque estas variables no afectan a la activación de la transición, sí pueden condicionar el valor que toman los términos que anotan los arcos de entrada y de salida de la transición. Cada vez que se verifica la regla anterior el razonador infiere un nuevo modo de transición identificado por `tmo(?_tra, ?_asl)`. A partir de este identificador se completarán las propiedades del modo de transición con las siguientes reglas:

```
hasTransition(tmo(?_tra, ?_asl), ?_tra) :-
  Mode(tmo(?_tra, ?_asl)).
```

```
hasAssignments(tmo(?_tra, ?_asl), ?_asl) :-
  Mode(tmo(?_tra, ?_asl)).
```

Las reglas anteriores sirven para inferir los modos de transición y sus propiedades pero no indican si un modo está activo en un determinado estado de la red. Para ello se usa la siguiente regla:

```
isEnabled(?_mod, ?_mar) :-
Marking(?_mar),
Mode(?_mod),
hasTransition(?_mod, ?_tra),
hasInputNodeList(?_tra, ?_pll),
  hasPlaceListEnoughTokens(?_pll, ?_mar, ?_mod).
```

En esta regla el predicado `hasPlaceListEnoughTokens` comprueba si la lista de plazas de entrada de la transición contiene las suficientes marcas para que el modo esté activo en el estado `?_mar`.

3.4.3.3. Reglas para el disparo de transiciones

Nuestra ontología también incluye reglas para el disparo de de modos de transición. Es necesario enfatizar dos de estas reglas en cuya parte consecuente están los predicados `doModeFiring` y `doModeListFiring` que posibilitan el disparo de un único modo o de una lista de modos de transición, respectivamente. Ambos predicados tienen tres parámetros y comparten los dos primeros: la instancia de ejecución de la HLPN y el estado actual de la red. Sin embargo, difieren en el tercer parámetro, ya que el primer predicado se refiere a un único modo de transición mientras que el segundo a una lista de modos. Este es el código asociado al primer predicado:

```
doModeFiring(?_exe, ?_mar, ?_mod) :-
  if   (hasCurrentMarking(?_exe, ?_mar))
  then (
  if   (isEnabled(?_mod, ?_mar))
  then (
    insert{Marking(mar(?_mar))},
    insert{Firing(fir(?_mar))},
    insert{hasSourceMarking(fir(?_mar), ?_mar)},
    insert{hasTargetMarking(fir(?_mar), mar(?_mar))},
    insert{hasModes(fir(?_mar), [?_mod])},
    delete{hasCurrentMarking(?_exe, ?_mar)},
    insert{hasCurrentMarking(?_exe, mar(?_mar))},
    hasFirings(?_exe, ?_ls1),
    append(?_ls1, [fir(mar)], ?_ls2),
    delete{hasFirings(?_exe, ?_ls1)},
    insert{hasFirings(?_exe, ?_ls2)})
  else (
    format("doModeFiring error:\n",
      [])@_prolog(format),
    format(" - The mode is not enabled
```

```
      in this marking\n", [])@_prolog(format)))
  else (
    format("doModeFiring error:\n",
      [])@_prolog(format),
    format(" - The net is on a different
      marking\n", [])@_prolog(format)).
```

La regla anterior comprueba que el estado actual se corresponde con el valor del parámetro *?_mar* y que el modo de transición está efectivamente activo en *?_mar*. En caso de cumplirse ambos requisitos, se crea el estado objetivo de la red y el disparo que propicia el cambio de estado y se completan los parámetros de ambos conceptos. Finalmente, se modifica la ejecución actualizando el estado actual de la red y la lista de disparos. Las marcas de cada una de las plazas para el nuevo estado de la red se infiere a través de la siguiente regla:

```
PlaceMarking(pma(?_tma, ?_pla, ?_tol)) :-
  hasSourceMarking(?_fir, ?_sma),
  hasTargetMarking(?_fir, ?_tma),
  hasModes(?_mol, ?_mol),
  hasMarking(?_pma, ?_sma),
  hasPlace(?_pma, ?_pla),
  hasTokens(?_pma, ?_ls1),
  getConsumedTokens(?_mol, ?_pla, ?_ls2),
  getProducedTokens(?_mol, ?_pla, ?_ls3),
  consumeTokens(?_ls2, ?_ls1, ?_ls4),
  produceTokens(?_ls3, ?_ls4, ?_tol).
```

Esta regla toma los modos de transición de cada disparo y consume/produce las correspondientes marcas en cada plaza, tal y como indica la regla de transición de las PNs. Finalmente, a partir del identificador del estado de cada plaza `pma(?_tma, ?_pla, ?_tol)`, donde `?_tma` indica el estado de la red, `?_pla` la plaza y `?_tol` la lista de marcas asociadas a ella, se completan las propiedades de las instancias del concepto `PlaceMarking`:

```
hasMarking(pma(?_tma, ?_pla, ?_tol), ?_tma) :-
  PlaceMarking(pma(?_tma, ?_pla, ?_tol)).

hasPlace(pma(?_tma, ?_pla, ?_tol), ?_pla) :-
  PlaceMarking(pma(?_tma, ?_pla, ?_tol)).

hasTokens(pma(?_tma, ?_pla, ?_tol), ?_tol) :-
  PlaceMarking(pma(?_tma, ?_pla, ?_tol)).
```

3.4.4. Compatibilidad con PNML

Uno de los objetivos de esta ontología fue su compatibilidad con los tres metamodelos especificados en el estándar internacional ISO/IEC 15909-2 (PNML) [145]. Nuestra ontología y PNML son soluciones diferentes aunque compartan el objetivo de representar PNs: PNML trata de conseguir un formato común de intercambio de fichero

para *todo* tipo de PNs y de editores gráficos, mientras que nuestra ontología tiene como propósito describir la semántica de los grafos y del modelo de ejecución de las HLPNs. Sin embargo, a pesar de estar pensadas para diferentes usos, es importante mantener la compatibilidad entre ambas especificaciones, especialmente considerando que PNML es el estándar *de facto* para el intercambio de PNs y que muchos editores lo utilizan en la actualidad para importar/exportar PNs. Como veremos en los siguientes apartados, la compatibilidad de nuestra ontología con PNML es total si descontamos *(i)* los elementos gráficos relacionados con los editores visuales y no incluidos en la ontología y *(ii)* aquellos aspectos relacionados con la ejecución de las PNs no contemplados por PNML.

3.4.4.1. Compatibilidad con el PNML Core Model

En este metamodelo se detalla *(i)* la estructura del grafo de una PN, *(ii)* la información gráfica asociada a esta estructura, y *(iii)* la información específica de los editores visuales encargados de representar estas redes. Es importante remarcar que tanto la información gráfica como la de los editores está fuera del ámbito de la ontología. Por ejemplo, conceptos como `ToolInfo` o `Graphics` no se consideran en la ontología, en la medida en que no describen ningún concepto de una PN. Teniendo esto en cuenta, la principales diferencias entre ambos modelos son:

- El concepto `PetriNetDoc` no se modela en la ontología porque un *documento* no pertenece al nivel conceptual de una HLPN.
- Los conceptos `Page`, `RefTrans` y `RefPlace` no se incluyen en la ontología, ya que están relacionados con los mecanismos de jerarquización y composición de HLPNs, los cuales no forman parte del estándar ISO/IEC 15909-1 [144]. Estos mecanismos de composición, no obstante, se tendrán en cuenta en la extensión jerárquica de la ontología de HLPNs (ver apartado 3.5). En cualquier caso, es importante mencionar que PNML no define explícitamente los mecanismos de composición ni su semántica: únicamente los utiliza para facilitar el visionado gráfico de las redes ocultando así parte de su complejidad.
- PNML Core Model está pensado para el intercambio de PNs entre editores gráficos, y por ello describe un grafo a partir de sus elementos visuales. Por ejemplo, un nodo será una etiqueta gráfica (concepto `Label`) con una posición en el editor, un color, un tipo de línea, etc. Sin embargo, la caracterización gráfica no debe describirse al mismo nivel que los componentes de una HLPN, ya que no forma parte de su especificación matemática tal y como queda recogida en el estándar ISO/IEC 15909-1. Por este motivo, la ontología no incluye elementos gráficos.

La Tabla 3.7 muestra las correspondencias entre PNML Core Model y la ontología de HLPNs: cada atributo y concepto de PNML se relaciona con su respectiva definición en la ontología (descrita en lógica de primer orden). Una plantilla XSLT ha sido

Tabla 3.7: Correspondencias entre atributos del PNML Core Model y la ontología de HLPNs

Concepto `PetriNet` del estándar ISO/IEC 15909-2		
Atributo	Correspondencia	Definición en F-Logic
id	El identificador de la instancia está implícitamente definido en el concepto `HLPN`.	`?_x:HLPN`
type	Este atributo se usa en PNML para especificar el tipo de PN modelada. PNML permite modelar, entre otros tipos, PNs de bajo nivel, redes plaza/transición y HLPNs. Ya que la ontología únicamente modela HLPNs, este atributo no tiene correspondencia.	−−
Concepto `Place` del estándar ISO/IEC 15909-2		
Atributo	Correspondencia	Definición en F-Logic
id	El identificador de la instancia está implícitamente definido en el concepto `Place`.	`Place(?_x)`
name	Al contrario que en PNML, este atributo se hereda de la superclase `Node`.	`dataProperty(hasName)` `domain(hasName, Node)`
Concepto `Transition` del estándar ISO/IEC 15909-2		
Atributo	Correspondencia	Definición en F-Logic
id	El identificador de la instancia está implícitamente definido en el concepto `Transition`.	`Transition(?_x)`
name	Al contrario que en PNML, este atributo se hereda de la superclase `Node`.	`dataProperty(hasName)` `domain(hasName, Node)`
Concepto `Arc` del estándar ISO/IEC 15909-2		
Atributo	Correspondencia	Definición en F-Logic
id	El identificador de la instancia está implícitamente definido en el concepto `Arc`.	`Arc(?_x)`
source	La diferencia entre ambos modelos es únicamente sintáctica. El nombre del atributo pasa de `source` a `hasSourceNode`.	`objectProperty(hasSourceNode)`
target	La diferencia entre ambos modelos es únicamente sintáctica. El nombre del atributo pasa de `target` a `hasTargetNode`.	`objectProperty(hasTargetNode)`

implementada para facilitar la traducción entre PNML y la implementación de la ontología en FLORA-2.

3.4.4.2. Compatibilidad con el PNML PT Model

Este metamodelo refina el PNML Core Model y facilita la representación de redes tipo Plaza/Transición (*Place/Transition nets*) añadiendo los conceptos para el marcado inicial y la anotación del grafo de la red, de forma que se permite asociar un número natural al estado inicial de las plazas y un número entero positivo a los arcos. Este metamodelo es un paso previo a la formulación del modelo de HLPN de PNML. Por ello no tiene sentido compararlo con la ontología ya los conceptos añadidos en esta capa son extendidos en el PNML HLPNG Model. Sin embargo, el aspecto más interesante de este metamodelo es que introduce axiomas en formato OCL en los diagramas UML. Específicamente, contiene dos axiomas que evitan que los arcos se

conecten a nodos del mismo tipo (plazas con plazas o transiciones con transiciones). Sendas restricciones también están capturadas en la ontología dentro de los axiomática presentada en la Tabla 3.1.

3.4.4.3. Compatibilidad con el PNML HLPNG Model

Este metamodelo extiende el PT Model con una firma sintáctica y con términos que permiten anotar las redes con una especificación algebraica. Las principales diferencias entre este modelo y la ontología descrita en este trabajo se resumen en los siguientes puntos:

- PNML no define el concepto de firma sintáctica (*signature)*. En PNML una firma es un paquete UML que contiene un conjunto de colores y operadores. Por ejemplo la firma lógica está compuesta por el color *boolean* y por los operadores *equality*, *inequality*, *and*, *or*, *not*, *imply*; la firma de los números enteros contiene los colores *integer*, *natural* y *positive*, y los operadores *addition*, *subtraction*, *multiplication*, *division*, *modulo*, *greaterthan*, *lessthan* y *lessthanorequal*. PNML predefine otros paquetes denominadas *CycleEnumerationSignature*, *StringSignature* y *ListSignature* que contienen los colores y operadores para tratar las enumeraciones, cadenas de texto y listas, respectivamente. Por lo tanto, en PNML una HLPN no está ligada explícitamente a una firma.
- La jerarquía de términos *no* se modela de la misma manera. La diferencia radica en la forma en que ambos modelos declaran el tipo de una plaza, la precondición de una transición y la anotación de los arcos. Estos componentes se modelan como cualquier otro elemento visual en PNML y para ello se extiende el concepto de etiqueta gráfica (concepto `Label`) para definir los colores (concepto `Type`), las condiciones de guardia (concepto `Condition`) y las anotaciones en los arcos (concepto `HLAnnotation`).
- La relación `hlinitialMarking` de PNML indica el estado inicial de las red a través de un conjunto de términos. Por el contrario, nuestra ontología sigue la definición del estándar ISO/IEC 15909-1 y modela el estado de la red como la agregación de los estados de las plazas. Es decir, mientras PNML únicamente considera el estado inicial, nuestra ontología, al modelar la dinámica de la red, considera todos los estados por los que pasa la ejecución de la red. Además, PNML también difiere en la representación del estado: lo representa mediante un término mientras que la ontología lo hace asociando a cada plaza el multiconjunto de valores (tokens) que contiene.
- La no existencia de un álgebra que dote de semántica operativa a la firma sintáctica es la principal diferencia entre el modelo PNML HLPNG y el modelo estático de la ontología de HLPNs. Claramente, PNML no cumple el estándar ISO/IEC 15909-1 en este punto, ya que no define las funciones que añaden semántica a las operaciones sintácticas. Esta carencia se debe a que PNML es un formato de intercambio entre herramientas de representación de PNs, y no

tiene en cuenta la semántica de ejecución de las HLPNs. Sin embargo, al no incluir esta parte del modelo, no permiten que dos herramientas puedan hacer simulaciones de una misma red con la *misma* semántica operacional. En cambio, la ontología sí proporciona esta semántica a través de las fórmulas asociadas a las funciones incluidas en el álgebra de la red.

PNML también incluye axiomas en el modelo HLPNG. Estos axiomas se usan para restringir la definición de los términos usados en la anotación de la red. La ontología propuesta en este trabajo contiene estos y más axiomas que restringen tanto la definición de la firma sintáctica, la jerarquía de términos como la estructura del grafo (además de los que restringen el álgebra, estado y ejecución de la red). A partir de este análisis, se puede apreciar cierta carencia axiomática en este metamodelo de PNML, con lo que es probable definir conceptos semánticamente incorrectos. Sin embargo, es justo señalar que muchos de los axiomas de la ontología que no están incluidos en PNML no podrían describirse a través de formato de representación OCL.

3.5. Ontología de HLPNs jerárquicas

Las HLPNs jerárquicas complementan a las HLPNs con capacidades de composición. Así, al igual que la introducción del color facilita la composición de PNs de bajo nivel que comparten cierta simetría, las redes jerárquicas facilitan la composición de HLPNs. A través de estos mecanismos de composición y estructuración de redes [120], se permite la creación de HLPNs complejas a partir de otras HLPNs más sencillas. Con este tipo de redes, los usuarios encargados de diseñar los procesos de negocio pueden construir sus sistemas usando una estrategia de arriba hacia abajo o de abajo hacia arriba.

3.5.1. Modelo de composición

La ontología de HLPNs jerárquicas captura la semántica de los mecanismos de fusión y sustitución de nodos definidos en las HLPNs [149]. El modelo que da soporte a estos mecanismos está representado en la red semántica de la Figura 3.17, en la que los elementos propios de esta ontología aparecen dentro del recuadro sombreado, mientras que los otros elementos se corresponden con conceptos de la ontología de HLPNs.

El concepto `HHLPN` se utiliza para definir una HLPN jerárquica a partir de un conjunto de páginas y de operaciones sobre esas páginas. La propiedad `hasName` identifica la red por medio de un nombre, mientras que la propiedad `hasDescription` proporciona un breve resumen de sus características más importantes:

```
subClassOf(HHLPN, Thing).

domain(hasName, HHLPN).
domain(hasDescription, HHLPN).
```

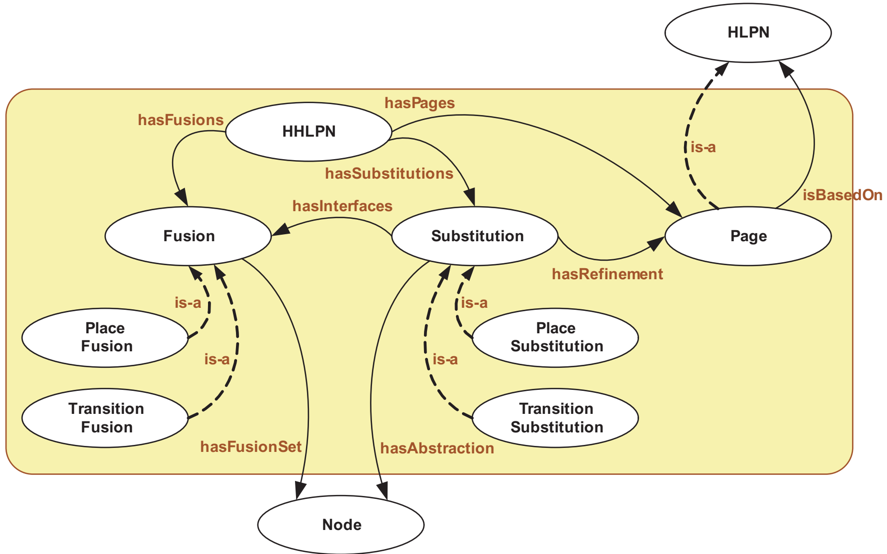


Figura 3.17: Red semántica de la ontología de redes de Petri jerárquicas

Las relaciones `hasFusions` y `hasSubstitutions` se refieren a las operaciones de composición con las que se crean las redes jerárquicas a través de fusiones y sustituciones de nodos, respectivamente:

```
objectProperty(hasFusions).
domain(hasFusions, HHLPN).
range(hasFusions, Fusion).

objectProperty(hasSubstitutions).
domain(hasSubstitutions, HHLPN).
range(hasSubstitutions, Substitution).
```

Finalmente, la relación `hasPages` se refiere a las páginas que componen una red jerárquica, que serán los elementos sobre los que se aplicarán las operaciones de composición anteriormente mencionadas:

```
objectProperty(hasPages).
domain(hasPages, HHLPN).
range(hasPages, Page).
minCardinality(hasPages, 1).
```

La clase `Page` permite representar una página, que es una copia de una HLPN no jerárquica que ha sido previamente definida a la que se referencia a través de la relación `isBasedOn`:

```
subClassOf(Page, HLPN).

objectProperty(isBasedOn).
domain(isBasedOn, Page).
range(isBasedOn, HLPN).
maxCardinality(hasPages, 1).
```

Es importante remarcar que una página comparte la misma firma sintáctica, variables, álgebra y anotaciones de la red de la que es copia: lo único que se copia es la estructura de la red imagen, ya que los mecanismos de composición necesitan operar sobre los nodos de las páginas. Esta duplicación es necesaria porque en una misma red jerárquica podrían existir varias páginas que se refieran a la *misma* HLPN. De este modo, si no se replicase la estructura de la red para cada página, sería imposible distinguir entre los nodos de las distintas páginas. En el ejemplo de fabricación descrito a lo largo de este capítulo es sencillo pensar en un escenario más complejo donde sea necesario replicar la estructura de la cadena de producción. Por ejemplo, si varias máquinas idénticas de ensamblado comparten la misma cinta de transporte, cada una de ellas se modelaría como una página que sería una imagen de la HLPN representada en la Figura 3.13.

Fusión de nodos. El primer mecanismo de composición de la ontología es la *fusión* de nodos (*Node-based folding*, en inglés), donde varias páginas se unen en un determinado nodo. Este tipo de composición horizontal también se utiliza como conveniencia gráfica para la representación del mismo nodo en diferentes posiciones del grafo [120]. En este sentido, en esta ontología se toma la definición de fusión más aceptada en la bibliografía científica [149, 140, 67, 68], y en la que, a través del concepto `Fusion`, se considera que un conjunto de nodos *pasan a ser* un mismo nodo:

```
subClassOf(Fusion, Thing).

objectProperty(hasFusionSet).
domain(hasFusionSet, Fusion).
range(hasFusionSet, Node).
minCardinality(hasFusionSet, 1).
```

La relación `hasFusionSet` establece los nodos que se consideran idénticos, pero no restringe que pertenezcan a la misma red. Por lo tanto, una fusión permite la creación de puntos de unión entre distintas redes (o en la misma) siempre y cuando las páginas pertenezcan a la misma red jerárquica. Dependiendo de si los nodos fusionados son plazas o transiciones, la fusión se especifica a través de los conceptos `PlaceFusion` o `PlaceTransition`, respectivamente:

```
subClassOf(PlaceFusion, Fusion).
subClassOf(TransitionFusion, Fusion).

allValuesFrom(hasFusionSet, PlaceFusion, Place).
allValuesFrom(hasFusionSet, TransitionFusion, Transition).

oneOf(Fusion,[PlaceFusion,TransitionFusion]).
disjoint(placeFusion, TransitionFusion).
```

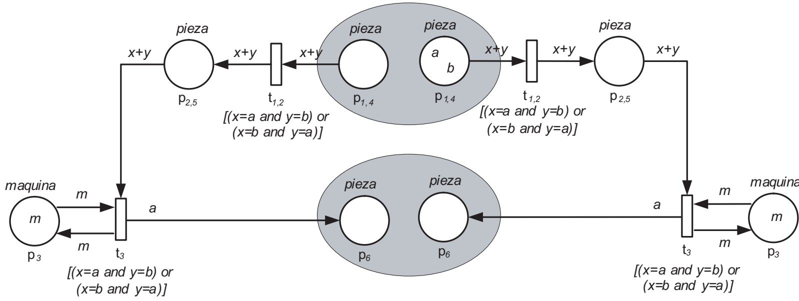


Figura 3.18: Las dos máquinas modeladas a través de la red de la Figura 3.13 comparten la cinta de transporte mediante dos fusiones, que se representan a través de elipses grises. En este caso, las plazas de entrada y salida de ambas máquinas representan la cinta de transporte.

Estos conceptos constituyen una partición exhaustiva y disjunta, y restringen que los nodos que forman parte de la fusión sean del mismo tipo. En el caso de la fusión de plazas además es, necesario tener en cuenta su tipo: ya que los nodos fusionados representan una única plaza, se requiere que dichas plazas tengan colores compatibles. En el caso de la fusión de transiciones no se añade ningún requisito adicional: únicamente hay que tener en cuenta que la precondición de la transición fusionada será la unión de las precondiciones de cada una de las transiciones que conforman la fusión.

Otro aspecto a tener en cuenta a la hora utilizar este mecanismo de composición es que las fusiones no comparten nodos. Esto significa que el mismo nodo no puede incluirse en dos fusiones distintas de la misma red jerárquica; si esto llegase a ocurrir, entonces sería necesario revisar el modelo o unir ambas fusiones. Siguiendo con el ejemplo de la planta de producción, para que dos máquinas, controladas mediante la red representada en la Figura 3.13, comparten la misma cinta de transporte (plazas $p_{1,4}$ y p_6), es necesario fusionar las plazas que representan dicha cinta. Por ejemplo, la Figura 3.18 representa la red resultante, en la que las plazas de entrada y salida de la red están fusionadas (elipse gris): las plazas $p_{1,4}$ de ambas redes representan una única plaza en el contexto de la red jerárquica. Además, el estado de la plaza fusionada contendrá los valores almacenados en ambas plazas; en este caso, a y b, que son los valores de la plaza $p_{1,4}$ de la red de la derecha.

La lista de axiomas que restringen la semántica de las fusiones está recogida en la Tabla 3.8.

Sustitución de nodos. El segundo mecanismo de composición es el concepto de *sustitución*, también conocido como refinado/abstracción jerárquica, que ha sido introducido para manejar la complejidad inherente de los modelos de HLPNs. A través de las sustituciones se consigue mitigar el efecto que produce el gran nivel de detalle de las HLPN y que en ocasiones no permite tener una perspectiva global del sistema diseñado ni de su comportamiento [120]. En este contexto, una sustitución relaciona

Tabla 3.8: Axiomas que restringen las fusiones de nodos

Los nodos fusionados pertenecen a páginas de la misma red jerárquica.
$\forall H, F, N\ HHLPN(H) \wedge Fusion(F) \wedge Node(N) \wedge hasFusions(H, F) \wedge hasFusionSet(F, N) \rightarrow \exists P\ Page(P) \wedge hasPages(H, P) \wedge hasNodes(P, N)$
Todos los nodos de una fusión de plazas son de tipo plaza.
$\forall F, P\ PlaceFusion(F) \wedge hasFusionSet(F, P) \rightarrow Place(P)$
Todos los nodos de una fusión de transiciones son de tipo transición.
$\forall F, T\ TransitionFusion(F) \wedge hasFusionSet(F, T) \rightarrow Transition(T)$
Las plazas fusionadas en una fusión de plazas son del mismo tipo.
$\forall F, P_1, P_2, S_1, S_2\ PlaceFusion(F) \wedge hasFusionSet(F, P_1) \wedge hasFusionSet(F, P_2) \wedge Place(P_1) \wedge Place(P_2) \wedge hasSort(P_1, S_1) \wedge hasSort(P_2, S_2) \rightarrow S_1 = S_2$
Los conjuntos de elementos fusionados son disjuntos en una HLPN jerárquica.
$\forall H, F_1, F_2, N_1, N_2\ HHLPN(H) \wedge Fusion(F_1) \wedge Fusion(F_2) \wedge hasFusions(H, F_1) \wedge hasFusions(H, F_2) \wedge hasFusionSet(F_1, N_1) \wedge hasFusionSet(F_2, N_2) \rightarrow \neg(N_1 = N_2)$

un nodo con una red más compleja, de modo que a través del nodo se abstrae el comportamiento de la red, permitiendo su descomposición de arriba hacia abajo y su construcción de abajo hacia arriba.

El concepto `Substitution` captura este mecanismo de composición vertical. La relación `hasAbstraction` apunta al nodo a sustituir, mientras que la relación `hasRefinement` contiene la página sustituta. Tanto el nodo como la página pertenecen a la propia red jerárquica, es decir, todas las páginas que participan en una sustitución tienen que incluirse en ella:

```
subClassOf(Substitution, Thing).

objectProperty(hasAbstraction).
domain(hasAbstraction, Substitution).
range(hasAbstraction, Node).
cardinality(hasAbstraction, 1).

objectProperty(hasRefinement).
domain(hasRefinement, Substitution).
range(hasRefinement, Page).
cardinality(hasRefinement, 1).
```

Al igual que en el caso de las fusiones, existen dos tipos de sustituciones dependiendo del nodo a reemplazar. El concepto `PlaceSubstitution` captura las sustituciones de plazas, mientras que el concepto `TransitionSubstitution` lo hace para las transiciones. Ambas clases constituyen la partición exhaustiva y disjunta de la jerarquía de sustituciones:

```
subClassOf(PlaceSubstitution, Substitution).
subClassOf(TransitionSubstitution, Substitution).

allValuesFrom(hasAbstraction, PlaceSubstitution, Place).
```

```
allValuesFrom(hasAbstraction, TransitionSubstitution, Transition).

oneOf(Substitution,[PlaceSubstitution,TransitionSubstitution]).
disjoint(PlaceSubstitution, TransitionSubstitution).
```

Una sustitución indica que un nodo de una red es reemplazado por otra red. Sin embargo, para llevar a cabo la composición es necesario añadir los puntos de unión entre las dos redes, es decir, establecer fusiones entre ambas redes. En el caso de la sustitución de una plaza la relación `hasInterfaces` contiene fusiones de transiciones, debido a que los arcos que apuntan a la plaza sustituida tienen como nodo origen a una transición. Para el caso de la sustitución de una transición el procedimiento es idéntico, pero lo que se fusiona son plazas:

```
objectProperty(hasInterfaces).
domain(hasInterfaces, Substitution).
range(hasInterfaces, Fusion).
minCardinality(hasInterfaces, 1).

allValuesFrom(hasInterfaces, PlaceSubstitution, TransitionFusion).
allValuesFrom(hasInterfaces, TransitionSubstitution, PlaceFusion).
```

Aplicado al ejemplo de fabricación, las sustituciones permiten establecer un comportamiento más preciso en alguna de las partes del ensamblaje de las piezas, por ejemplo en su preparación. En este caso, tal y como se muestra en la Figura 3.19, se realiza un proceso en paralelo donde operarios de la planta de fabricación seleccionan una pieza a y otra b. Por una parte, la pieza de tipo a constituye el bloque a ensamblar y por ello se le aplica cola en toda la superficie de contacto que la unirá a la pieza b. Por otra parte, a la pieza b se le aplica cola en los taladros y se le añade el tojino que la unirá a la pieza a. Sustituyendo la transición $t_{1,2}$ por la página *preparación* se consigue el efecto deseado y se enriquece el modelo, pero manteniendo la visión global del sistema. En la figura no se han dibujado las fusiones que permiten unir ambas redes debido a cuestiones de claridad, pero en la definición de esta red existen dos fusiones: entre las plazas $p_{1,4}$ y *piezas a encolar* y entre las plazas $p_{2,5}$ y *piezas preparadas para encolado*.

Finalmente, es necesario tener en cuenta una última restricción a la hora de definir una sustitución: no se admiten referencias recursivas entre redes, es decir, no pueden existir dos sustituciones en las que se permita sustituir un determinado nodo de la red x por la red y y un nodo de la red y por la red x. Esta restricción no sólo se aplica a pares de sustituciones, sino también a su encadenamiento.

La tabla 3.9 describe la restricción anterior y otros axiomas aplicables a las sustituciones de nodos.

3.5.2. Morfismo con red aplanada de alto nivel

En el apartado anterior se han presentado los conceptos que capturan la semántica del modelo estático de las redes jerárquicas. Sin embargo, a través de esa taxonomía no

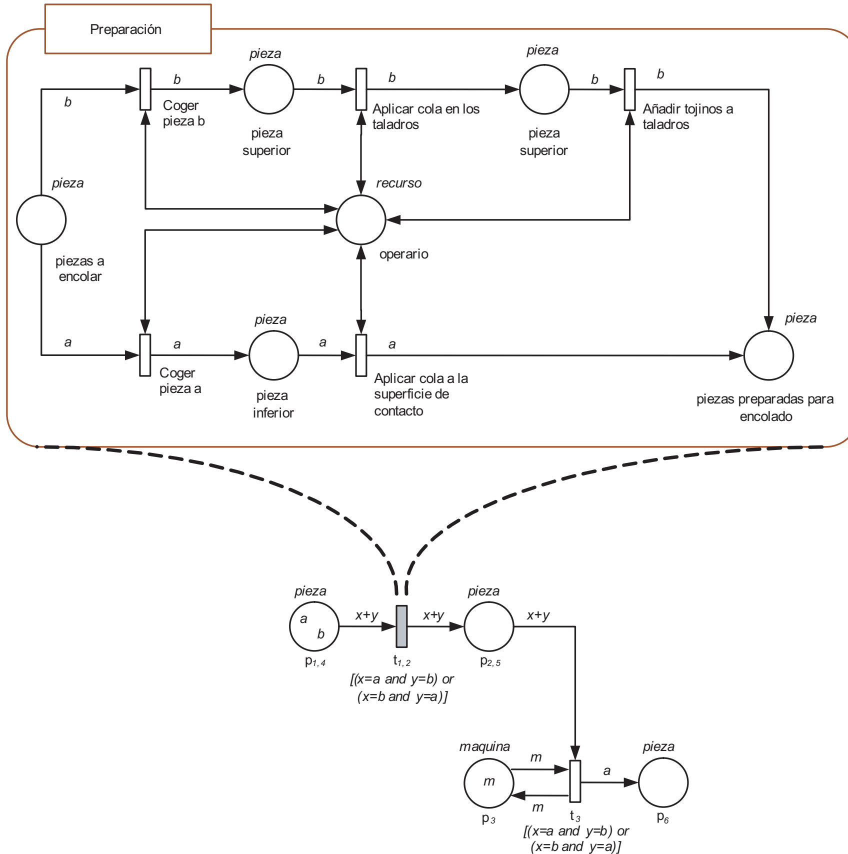


Figura 3.19: Refinamiento del proceso de preparación de las piezas para su posterior procesado. En este caso, la transición $t_{1,2}$ es sustituida por la página *preparación* que captura con más detalle el proceso.

es posible representar su ejecución: en ella no se aporta ningún concepto que permita saber el comportamiento de sus componentes durante la ejecución. Sin embargo, es necesario modelar dicha semántica para poder ejecutar redes jerárquicas.

Existen dos aproximaciones diferentes que permiten el modelado de la semántica operacional de las HLPNs jerárquicas. La primera consiste en crear una semántica de ejecución que sea específica para este tipo de redes. Consistiría en definir, al igual que para las HLPNs, un modelo de estado y otro de ejecución. Así, el estado de una red jerárquica podría consistir en la combinación de los estados individuales de las páginas que la conforman. El problema de esta aproximación es la sincronización de las plazas que están fusionadas: cuando se elimina o se crea una marca en una plaza que pertenece a una fusión es necesario sincronizar las demás plazas que pertenecen a la fusión. Si estas plazas se encuentran en una red diferente, no existe una forma

Tabla 3.9: Axiomas que restringen a las sustituciones de nodos

Las páginas que sustituyen algún nodo pertenecen a la red jerárquica.
$\forall S, H, P\ Substitution(S) \wedge HHLPN(H) \wedge hasRefinement(S, P) \wedge hasSubstitutions(H, S) \rightarrow hasPages(H, P)$
El nodo sustituido en una sustitución pertenece a una página de la red jerárquica.
$\forall S, H, P\ Substitution(S) \wedge HHLPN(H) \wedge hasAbstraction(S, N) \wedge hasPages(H, P) \wedge hasSubstitutions(H, S) \rightarrow Page(P) \wedge hasNodes(P, N)$
El nodo sustituido en una sustitución de plaza es de tipo plaza.
$\forall N, S\ Node(N) \wedge PlaceSubstitution(S) \wedge hasAbstraction(S, N) \rightarrow Place(N)$
El nodo sustituido en una sustitución de transición es de tipo transición.
$\forall N, S\ Node(N) \wedge TransitionSubstitution(S) \wedge hasAbstraction(S, N) \rightarrow Transition(N)$
Las sustituciones de plazas están relacionadas con fusiones de transiciones.
$\forall S, F\ PlaceSubstitution(S) \wedge Fusion(F) \wedge hasInterfaces(S, F) \rightarrow TransitionFusion(F)$
Las sustituciones de transiciones están relacionadas con fusiones de plazas.
$\forall S, F\ TransitionSubstitution(S) \wedge Fusion(F) \wedge hasInterfaces(S, F) \rightarrow PlaceFusion(F)$
Las entradas del nodo sustituido pertenecen a una de las fusiones de la sustitución.
$\forall S, A, N_1, N_2\ NodeSubstitution(S) \wedge Arc(A) \wedge hasSourceNode(A, N_1) \wedge hasTargetNode(A, N_2) \wedge hasAbstraction(S, N_2) \rightarrow \exists F\ hasInterfaces(S, F) \wedge hasFusionSet(F, N_1)$
Las salidas del nodo sustituido pertenecen a una de las fusiones de la sustitución.
$\forall S, A, N_1, N_2\ NodeSubstitution(S) \wedge Arc(A) \wedge hasSourceNode(A, N_1) \wedge hasTargetNode(A, N_2) \wedge hasAbstraction(S, N_1) \rightarrow \exists F\ hasInterfaces(S, F) \wedge hasFusionSet(F, N_2)$
No se permiten referencias recursivas en la definición de las sustituciones.
$\forall S, A, N_1, N_2\ HHLPN(H) \wedge Page(P_1) \wedge Page(P_2) \wedge hasPages(H, P_1) \wedge hasPages(H, P_2) \rightarrow \neg inTheSamePath(P_1, P_2)$

directa de indicar que su estado ha cambiado, es decir, que ya no contienen las mismas marcas. Para resolver este problema, una posibilidad sería extender la semántica de las HLPNs para dar soporte a *disparos virtuales* que apunten a disparos de otras redes. Sin embargo, esta semántica no cumpliría con alguno de los axiomas definidos para las HLPNs. Por ejemplo se dejaría de cumplir que el cambio de estado está motivado por el disparo de una transición, es decir, que una transición de la red elimina marcas de las plazas de entrada y genera nuevas marcas en las plazas de salida. Con la semántica de ejecución anterior podrían desaparecer marcas en las plazas sin que se haya producido el disparo de una de las transiciones de la red, sino simplemente porque una transición de otra red se disparó y eliminó la marca de una plaza perteneciente a una fusión. Por ello, hemos optado por seguir una aproximación distinta, que consiste en traducir la estructura de la red jerárquica a una única HLPN, es decir, en *aplanar la red jerárquica*. Con esta solución se utilizaría la semántica operacional de las HLPNs para ejecutar a las redes jerárquicas.

El modelo que asocia una red jerárquica con su HLPN equivalente está represen-

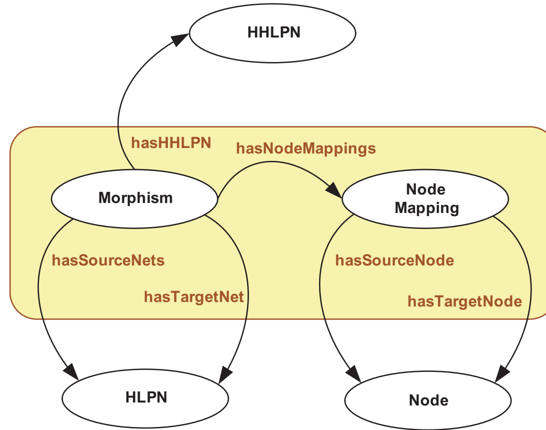


Figura 3.20: Red semántica del morfismo que se define entre la red jerárquica compuesta de múltiples páginas y su la correspondiente red aplanada

tado en la Figura 3.20. Este modelo crea un morfismo entre ambas redes, que como se verá más adelante, preserva las características de la red jerárquica en la aplanada. Así, el concepto `Morphism` relaciona las páginas de la red jerárquica con la HLPN aplanada:

```
subClassOf(Morphism, Thing).

objectProperty(hasHHLPN).
domain(hasHHLPN, Morphism).
range(hasHHLPN, HHLPN).
cardinality(hasHHLPN, 1).
```

La relación `hasSourceNets` apunta al conjunto de páginas de la red jerárquica que es necesario aplanar, mientras que la relación `hasTargetNet` se refiere a la HLPN que es el resultado del proceso de aplanamiento:

```
objectProperty(hasSourceNets).
domain(hasSourceNets, Morphism).
range(hasSourceNets, HLPN).
minCardinality(hasSourceNets, 1).

objectProperty(hasTargetNet).
domain(hasTargetNet, Morphism).
range(hasTargetNet, HLPN).
cardinality(hasTargetNet, 1).
```

La HLPN puede verse como la *unión* de las distintas páginas que conforman la red jerárquica. Por ello, en el proceso de aplanamiento se debe preservar la firma, variables, álgebra, anotaciones y estructura de cada una de sus páginas, de modo que la HLPN aplanada será la unión de todos estos componentes.

Dado el grafo de dos redes de alto nivel, $NG_1 = (P_1, T_1, F_1)$ y $NG_2 = (P_2, T_2, F_2)$, la correspondencia entre ellas puede definirse como la función $\varphi : P_1 \cup T_1 \rightarrow P_2 \cup T_2$, que relaciona los nodos (plazas y transiciones) de ambas redes. El concepto `NodeMapping` representa la aplicación de la función φ, es decir, conecta un nodo de una página con uno de la red aplanada:

```
subClassOf(NodeMapping, Thing).

objectProperty(hasSourceNode).
domain(hasSourceNode, NodeMapping).
range(hasSourceNode, Node).
cardinality(hasSourceNode, 1).

objectProperty(hasTargetNode).
domain(hasTargetNode, NodeMapping).
range(hasTargetNode, Node).
cardinality(hasTargetNode, 1).
```

La relación `hasNodeMappings` se refiere al conjunto de correspondencias definidas entre los nodos de las páginas de la red jerárquica y los de la HLPN aplanada:

```
objectProperty(hasNodeMappings).
domain(hasNodeMappings, Morphism).
range(hasNodeMappings, NodeMapping).
```

Sin embargo, para verificar que la red aplanada resultante mantiene la estructura y propiedades de la red jerárquica son necesarios un conjunto de axiomas adicionales. Así, se debe verificar que *(i)* la correspondencia entre dos plazas mantiene el color en la plaza de la red aplanada, *(ii)* se conserva la precondición de las transiciones, y *(iii)* se conservan los arcos, es decir, que los arcos se mantienen en las dos redes: $\forall x, y \in P_1 \cup T_1 \ (x, y) \in F_1 \Rightarrow (\varphi(x), \varphi(y)) \in F_2$.

El morfismo también tiene que verificar que los mecanismos de composición se traducen correctamente. Así, una fusión entre un conjunto de nodos se representa en la red aplanada como un único nodo. Por ello, cuando dos nodos x e y pertenecen a una misma fusión, entonces la correspondencia definida para ellos toma el mismo valor, es decir, $\varphi(x) = \varphi(y) = z$. Esta propiedad se aplica tanto a la fusión de plazas como de transiciones, en la cual también debe conservar las precondiciones en la HLPN aplanada. En ese caso, si se fusionan dos transiciones t_1 y t_2 con precondiciones P_1 y P_2, el valor de la precondición de la transición imagen será $P_1 \cup P_2$.

Como se comentó anteriormente, aunque la sustitución de nodos es un mecanismo de composición vertical, las fusiones que surgen de aplicar una sustitución son las encargadas de conectar las redes. Así, si un nodo x es sustituido, entonces desaparecerá de la HLPN aplanada, $\varphi(x) = \phi$, y en su lugar la red que lo sustituye estará conectada al resto de redes mediante fusiones. Por ejemplo, la Figura 3.21 muestra la red aplanada que es imagen de la red jerárquica representada en la Figura 3.19. En este caso, la transición $t_{1,2}$ desaparece de la red aplanada y es sustituida por la página *preparación*. Además, las fusiones definidas entre las plazas $p_{1,4}$ y *piezas a encolar* y

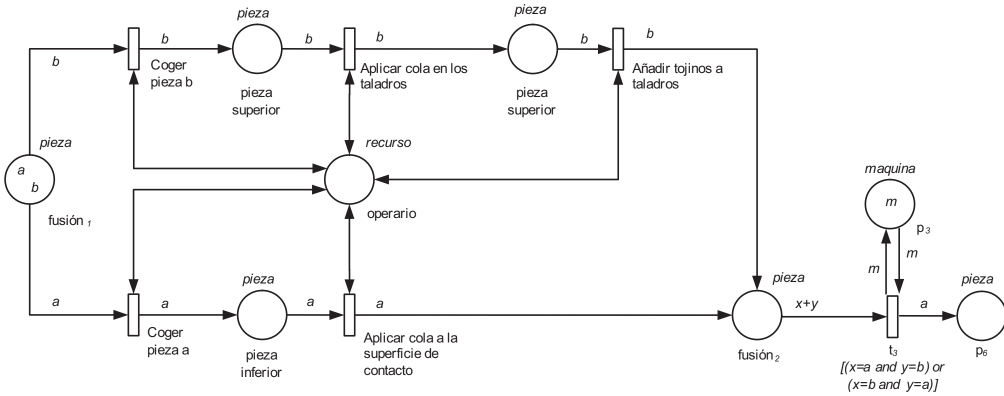


Figura 3.21: PN de alto nivel aplanada resultante del mecanismo de sustitución representado en la Figura 3.19

entre las plazas $p_{2,5}$ y *piezas preparadas para encolado* dan lugar a las nuevas plazas $fusion_1$ y $fusion_2$, respectivamente.

Las tablas 3.10 y 3.11 muestran el conjunto de axiomas adicionales necesarios para verificar que la semántica definida en la red jerárquica se mantiene en la HLPN aplanada.

3.5.3. Reglas

En la ontología de redes jerárquicas también se definen *(i)* un conjunto de reglas que facilitan la creación, modificación y borrado de los elementos de este tipo de redes, y *(ii)* reglas que permiten el aplanamiento de una red jerárquica para generar como resultado una HLPN equivalente. Esta regla automatiza la creación de la red aplanada y el establecimiento de las correspondencias definidas por el morfismo entre ambas redes. El siguiente código muestra una parte de estas reglas:

```
transformHHlpn2Hlpn(?_hhn, ?_fla) :-
  HHLPN(?_hhn),
  hasPageList(?_hhn, ?_pal),
  createHlpn(?_fla),
  setProperty(?_fla, HLPN, hasName, ?_hhn),
  setProperty(?_fla, HLPN, hasDescription, ?_hhn),
  createMorphism(?_mor),
  setPropertyList(?_mor, morphism, hasSourceNets, ?_pal),
  setProperty(?_mor, morphism, hasTargetNet, ?_fla),
  setProperty(?_hhn, HHLPN, hasMorphism, ?_mor),
  associateSignature(?_hhn, ?_fla),
  associateVariables(?_hhn, ?_fla),
  associateStructure(?_hhn, ?_fla).
```

Tabla 3.10: Axiomas que restringen el morfismo entre redes (I)

Conservación del color en la correspondencia entre plazas.
$\forall M, P_1, P_2, S_1, S_2\ NodeMapping(M) \wedge Place(P_1) \wedge Place(P_2) \wedge hasSourceNode(M, P_1) \wedge hasTargetNode(M, P_2) \wedge hasSort(P_1, S_1) \wedge hasSort(P_2, S_2) \rightarrow S_1 = S_2$
Conservación de la precondición en la correspondencia entre transiciones. Si la transición pertenece a una fusión, entonces la nueva precondición es la conjunción de las condiciones de los elementos de la fusión.
$\forall M, T_1, T_2, S_1, S_2\ NodeMapping(M) \wedge Transition(T_1) \wedge Transition(T_2) \wedge hasSourceNode(M, T_1) \wedge hasTargetNode(M, T_2) \wedge hasGuard(T_1, G_1) \wedge hasGuard(T_2, G_2) \rightarrow isGuardIncluded(G_1, G_2)$
Los nodos de una fusión se corresponden con un único nodo de la red aplanada.
$\forall M_1, M_2, N_1, N_2, F\ NodeMapping(M_1) \wedge NodeMapping(M_2) \wedge Fusion(F) \wedge hasSourceNode(M_1, N_1) \wedge hasSourceNode(M_2, N_2) \wedge hasFusionSet(F, N_1) \wedge hasFusionSet(F, N_2) \rightarrow \exists N_3\ hasTargetNode(M_1, N_3) \wedge hasYargetNode(M_2, N_3)$
Los nodos sustituidos no están incluidos en la red aplanada.
$\forall H, \varphi, S, M, N\ HHLPN(H) \wedge Morphism(\varphi) \wedge substitutuion(S) \wedge NodeMapping(M) \wedge Node(N) \wedge hasSubstitutions(H, S) \wedge hasMorphism(\varphi, H) \wedge hasAbstraction(S, N) \wedge hasNodeMappings(\varphi, M) \rightarrow \neg hasSourceNode(M, N)$

Como se puede ver, el predicado `transformHHlpn2Hlpn` se encarga de *(i)* crear el identificador de la HLPN en la variable `?_fla` (`createHlpn`), *(ii)* asignar el nombre y la descripción a `?_fla`, *(iii)* crear el identificador del morfismo en la variable `?_mor` (`createMorphism`), *(iv)* establecer el morfismo entre la red jerárquica `?_hhn` y la HLPN aplanada `?_fla`, y finalmente *(v)* definir la firma, las variables y la estructura a la HLPN apuntada por `?_fla`. Para establecer esta estructura a partir de la red jerárquica se hace uso del predicado `associateStructure`:

```
associateStructure(?_hhn, ?_fla) :-
  HHLPN(?_hhn),
  HLPN(?_fla),
  hasMorphism(?_hhn, ?_mor),
  hasFusionList(?_hhn, ?_ful),
  hasPageList(?_hhn, ?_pal),
  copyNodes(?_mor, ?_pal),
  copyFusionNodes(?_mor, ?_ful),
  copyArcs(?_mor, ?_pal).
```

En esta regla se muestran los dos pasos que permiten aplanar la estructura jerárquica en una HLPN. El primer paso consiste en copiar todos los nodos de cada una de las páginas pertenecientes a la red jerárquica `?_hhn` en la HLPN aplanada `?_fla`. Para ello se aplican los siguientes criterios:

- Los nodos que no pertenecen a ninguna fusión ni sustitución se copian directamente (`copyNodes`).

Tabla 3.11: Axiomas que restringen el morfismo entre redes (II)

La red aplanada contiene la firma de cada una de las páginas de la red jerárquicas.
$\forall H, \varphi, N_1, N_2, S_1, S_2\ HHLPN(H) \wedge Morphism(\varphi) \wedge hasMorphism(\varphi, H) \wedge hasSourceNets(\varphi, N_1) \wedge hasTargetNet(\varphi, N_2) \wedge hasSignature(N_1, S_1) \wedge hasSignature(N_2, S_2) \rightarrow isSignatureIncluded(S_1, S_2)$
La red aplanada contiene el álgebra de cada una de las páginas de la red jerárquicas.
$\forall H, \varphi, N_1, N_2, A_1, A_2\ HHLPN(H) \wedge Morphism(\varphi) \wedge hasMorphism(\varphi, H) \wedge hasSourceNets(\varphi, N_1) \wedge hasTargetNet(\varphi, N_2) \wedge hasAlgebra(N_1, A_1) \wedge hasAlgebra(N_2, A_2) \rightarrow isAlgebraIncluded(A_1, A_2)$
La red aplanada contiene las variables definidas en las páginas de la red jerárquicas.
$\forall H, \varphi, N_1, N_2, V_1, V_2\ HHLPN(H) \wedge Morphism(\varphi) \wedge hasMorphism(\varphi, H) \wedge hasSourceNets(\varphi, N_1) \wedge hasTargetNet(\varphi, N_2) \wedge hasVariables(N_1, V_1) \rightarrow hasVariables(N_2, V_1)$
Una plaza de la red jerárquica se corresponden con una plaza de la red aplanada, siempre y cuando dicha plaza no haya sido sustituida.
$\forall H, \varphi, M, P_1, P_2, S\ HHLPN(H) \wedge Morphism(\varphi) \wedge NodeMapping(M) \wedge Place(P_1) \wedge Place(P_2) \wedge hasMorphism(\varphi, H) \wedge hasNodeMappings(\varphi, M) \wedge hasSourceNode(M, P_1) \wedge hasTargetNode(M, P_2) \wedge \neg hasAbstraction(S, P_1) \rightarrow \exists N_1, N_2\ HLPN(N_1) \wedge HLPN(N_2) \wedge hasSourceNets(\varphi, N_1) \wedge hasTargetNet(\varphi, N_2) \wedge hasNodes(N_1, P_1) \wedge hasNodes(N_2, P_2)$
Una transición de la red jerárquica se corresponde con una transición de la red aplanada, siempre y cuando dicha transición no haya sido sustituida.
$\forall H, \varphi, M, T_1, T_2, S\ HHLPN(H) \wedge Morphism(\varphi) \wedge NodeMapping(M) \wedge Transition(T_1) \wedge Transition(T_2) \wedge hasMorphism(\varphi, H) \wedge hasNodeMappings(\varphi, M) \wedge hasSourceNode(M, T_1) \wedge hasTargetNode(M, T_2) \wedge \neg hasAbstraction(S, T_1) \rightarrow \exists N_1, N_2\ HLPN(N_1) \wedge HLPN(N_2) \wedge hasSourceNets(\varphi, N_1) \wedge hasTargetNet(\varphi, N_2) \wedge hasNodes(N_1, T_1) \wedge hasNodes(N_2, T_2)$

- Los nodos que pertenecen a una fusión dan lugar a un único nodo en la red aplanada y a la definición del conjunto de correspondencias dentro del morfismo entre ambas redes (`copyFusionNodes`).

- Los nodos sustituidos no se copian a la red plana.

El segundo paso consiste en la creación de los arcos (`createArcs`), que es sin duda el punto más complicado del proceso de aplanamiento anterior y que requiere copiar los arcos en función de las correspondencias establecidas entre los nodos durante el paso anterior.

Otra regla a destacar es la que facilita la *copia* de una HLPN para su reutilización en una red jerárquica (`createPage`). Es necesario recordar que las páginas que componen la red jerárquica no pueden compartir estructura a nivel de instanciación (por ejemplo, un mismo nodo no puede estar en dos páginas a la vez). Para resolver este

problema, cuando un usuario quiere reutilizar una HLPN, basta con invocar al predicado `createPage` para que la regla encargada de inferirlo realice el proceso correcto de replicación:

```
createPage(?_net, ?_oid) :-
  HLPN(?_net),
  hasName(?_net, ?_nam),
  hasDescription(?_net, ?_des),
  hasSignature(?_net, ?_sig),
  hasAlgebra(?_net, ?_alg),
  hasVariableList(?_net, ?_val),
  createObject(Page, ?_oid),
  updatePage(?_net, ?_oid),
  setProperty(?_oid, Page, name, ?_nam),
  setProperty(?_oid, Page, description, ?_des),
  setProperty(?_oid, Page, Signature, ?_sig),
  setProperty(?_oid, Page, Algebra, ?_alg),
  setPropertyList(?_oid, Page, variables, ?_val),
  copyNetStructure(?_net, ?_oid).
```

3.6. Conclusiones

En este capítulo hemos presentado dos metamodelos para describir y componer HLPNs. El primer metamodelo se centra en el modelado semántico de las HLPNs, que extienden el formalismo básico con color y anotación, para así simplificar las redes y evitar la explosión de nodos. El metamodelo define conceptos como nodo, arco, transición y plaza, que posibilitan describir su estructura de grafo bipartito y dirigido. Además, también incluye todos los elementos que permiten anotar algebraicamente estos grafos.

En lo que se refiere a la anotación, a diferencia de otras ontologías de HLPNs y de otros metamodelos como el descrito en el estándar ISO/IEC 15909-2, el modelo estático propuesto separa *explícitamente* la firma sintáctica de la red de su interpretación semántica, es decir, de su álgebra. Además de facilitar la ejecución, esta separación incrementa la capacidad de reutilización de las HLPNs: una red anotada sintácticamente puede ser interpretada por varias álgebras y, por lo tanto, ser reutilizada estableciendo las correspondencias entre la firma y el álgebra. La Figura 3.22 muestra una HLPN inicialmente creada para el cálculo del precio de fabricación de muebles a medida. Un experto en la fabricación de piensos compuestos podría considerar que esta red se adapta a su dominio y querer reutilizarla directamente. La adaptación consistiría en sustituir el álgebra utilizada en la fabricación de muebles por el álgebra para la fabricación de piensos. En la figura puede verse de forma simplificada cómo se realizaría esta adaptación: se asocian los tipos de soporte y las funciones del álgebra de piensos con los colores y operadores que anotan la HLPN. Por ejemplo, el soporte *RequerimientosAnimal* interpreta el color *Requerimientos*, la función *RevisarComposición* (a través de su correspondiente expresión) interpreta al operador *RevisarDiseño*, etc.

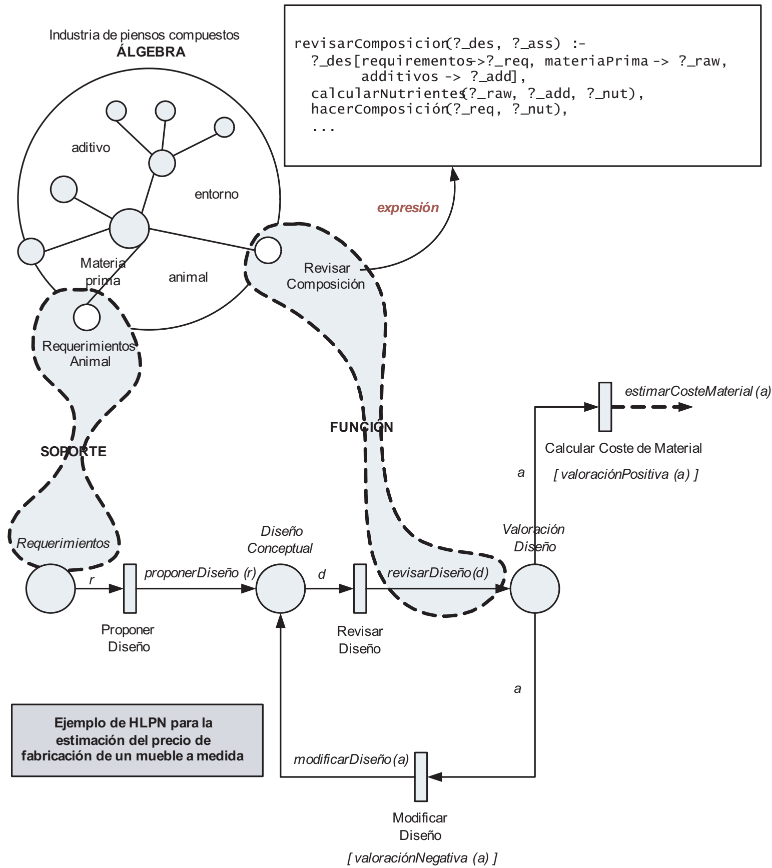


Figura 3.22: Ejemplo de HLPN inicialmente creada para estimar el coste de fabricación de un mueble pero que se puede reutilizar en el dominio de la fabricación de piensos compuestos

Otra diferencia de nuestro metamodelo respecto a las demás propuestas existentes es su capacidad para capturar la semántica del modelo de estado y de ejecución de las HLPNs. Esta capacidad permite que dos o más agentes compartan la información obtenida durante la ejecución de una HLPN. Por ejemplo, la Figura 3.23 plantea dos situaciones que enfatizan la importancia de esta información. La parte izquierda de la figura muestra una situación donde un conjunto de agentes comparten la misma instancia de ejecución de la HLPN. Esta situación puede darse en el mundo real, por ejemplo, en un sistema donde cada una de las transiciones a ejecutar esté a cargo de un agente. Como se verá en el Capítulo 6, esta configuración es similar a la usada por muchos de los sistemas de gestión de WFs. La segunda situación, mostrada en la parte derecha de la figura, ilustra una ejecución coordinada entre agentes: cada agente realiza una operación y delega la ejecución de la siguiente operación a otro agente.

Tabla 3.12: Capacidades de representación de las distintas ontologías de HLPNs: 1-Ontología de HLPNs [266], 2-PNML [145], 3-Gasevic y Devedzic [112], 4-Songfeng [236]. Los signos '+' indican un soporte directo, los '+/-' un soporte parcial o incompleto y los '-' que no soporta la característica indicada.

Característica	1	2	3	4
Modelo estático	+	+	+	+
Modelo de estado	+	-	-	-
Modelo de ejecución	+	-	-	-
Separación entre firma sintáctica y álgebra	+	-	-	-
Axiomática	+	+/-	+/-	-

Esta configuración suele darse en entornos distribuidos y especializados en los que cada nodo realiza parte de una ejecución global o en arquitecturas tolerantes a fallos, donde el agente recupera el estado de la ejecución en el punto crítico. En cualquiera de las situaciones anteriores, los agentes podrían ser ligeros, ya que la funcionalidad que tendrían que ejecutar estaría implementada mayoritariamente en el álgebra de la ontología de HLPN.

Para completar la ontología, el metamodelo incorpora un conjunto de axiomas que restringen la forma en la que se crean las instancias de la ontología. La mayor parte de las ontologías de HLPNs analizadas no incorporan ninguna axiomática y cuando lo hacen se limitan a expresar axiomas de subclases o de cardinalidad. En el presente metamodelo se han creado más de 20 axiomas que controlan que el grafo sea bipartito y dirigido, que la anotación esté correctamente construida, que la firma sintáctica y el álgebra interpreten correctamente la anotación, y que la semántica de ejecución se verifique en cada cambio de estado.

Además, en este capítulo describimos también una capa de reglas que facilita el uso del metamodelo y que constituirá la base del motor de ejecución de HLPNs descrito en el Capítulo 6. Entre otras funcionalidades, estas reglas permiten evaluar los términos que anotan las redes, calcular los modos en los que las transiciones están activas e incluso ejecutar un disparo.

Una comparativa más detallada entre las distintas ontologías de HLPNs se muestra en la Tabla 3.12.

Ontología de redes jerárquicas. El segundo metamodelo trata el modelado de las HLPNs jerárquicas en el que se traslada la especificación matemática descrita en [149] a un modelo de ontologías. Aunque existen otros modelos de redes jerárquicas y de composición de PNs, esta especificación es la más aceptada por la comunidad científica y en ella se basará el futuro estándar ISO/IEC 15909-3.

El metamodelo presentado permite construir redes jerárquicas a partir de HLPNs más simples que están unidas a través de dos mecanismos de composición: las fusiones y las sustituciones. El primer mecanismo de composición, denominado *fusión*, permite indicar que un nodo es la unión de un conjunto de nodos. Es un mecanismo de composición horizontal que permite unir distintas redes en un único nodo. El segundo

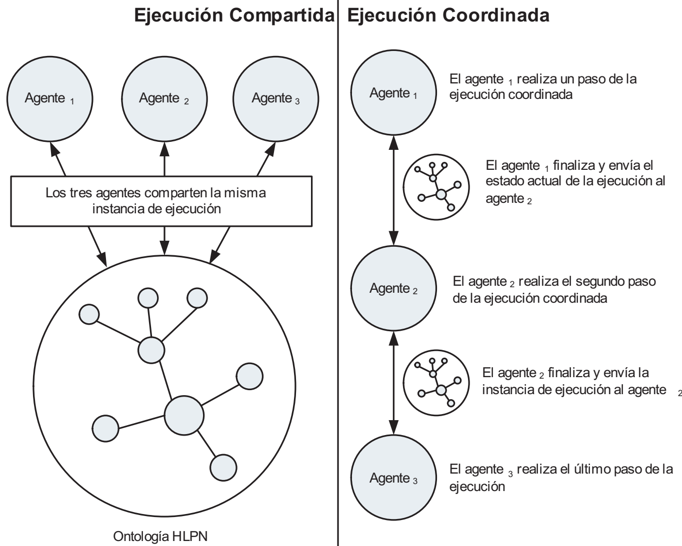


Figura 3.23: Los agentes pueden compartir la información dinámica generada durante la ejecución de una HLPN

mecanismo de composición se denomina *sustitución* y permite indicar que un nodo va a ser sustituido por otra HLPN. Aunque las redes jerárquicas aportan la semántica de composición de las HLPNs, no tienen una semántica formal de ejecución. Sin embargo, una red jerárquica puede transformarse en una HLPN equivalente y ejecutarse bajo este formalismo, para lo cual el metamodelo incorpora, a través de un morfismo entre redes, las correspondencias entre las HLPNs que componen la red jerárquica y la HLPN equivalente. Este metamodelo también contiene un conjunto de axiomas que restringen la semántica de los conceptos jerárquicos, y al igual que en el caso de las HLPNs, también tiene asociado un conjunto de reglas que facilitan el manejo de estas redes.

Finalmente, tal y como se comentó en el Capítulo 1, el principal problema de los marcos de modelado del conocimiento cuando se aplican a los WFs es su incapacidad para modelar adecuadamente el control. Vimos que la mayor parte de las aproximaciones no usan un formalismo de representación de procesos y, por lo tanto, carecen de técnicas para modelarlos adecuadamente. La conclusión es clara: *si se quieren modelar los WFs desde la perspectiva del conocimiento es necesario basar su representación en un lenguaje de modelado de WFs*. Tras el análisis realizado en el Capítulo 2, se concluyó que las HLPNs son uno de los formalismos más empleados en la bibliografía y en los sistemas comerciales relacionados con WFs: su naturaleza gráfica y su expresividad las convierten en la herramienta perfecta para describir la estructura de cualquier proceso y por ello las seleccionamos para modelar nuestros WFs. En este capítulo hemos presentado una ontología de HLPNs que permitirá establecer las bases para la representación de procesos dentro del metamodelo de WFs,

que describiremos en el siguiente capítulo. En este punto uno podría preguntarse: ¿por qué crear una ontología para ello? Ciertamente, se podría pensar que es suficiente con las técnicas de análisis y verificación de propiedades disponibles en la bibliografía de las HLPNs. Sin embargo, consideramos que era necesario proporcionar una base semántica sobre la cual razonar acerca de las características y comportamiento de los WFs, y precisamente las ontologías cubren este punto. Debido a esta capacidad, y a que proporcionan un modelo procesable, nuestro metamodelo permitirá abordar en un futuro tareas como el descubrimiento o la composición de WFs, diferenciándose en este punto de la mayoría de los productos comerciales existentes en la actualidad.

4

Metamodelo del marco conceptual de flujos de trabajo

En este capítulo se describe el metamodelo que da soporte al modelado de WFs desde una perspectiva semántica y orientada a la reutilización de conocimiento [271, 267]. Este metamodelo está construido a partir de la unión de tres metamodelos, cada uno de los cuales da soporte a una parte del diseño de los WFs. El primer pilar, el metamodelo UPML (del inglés *Unified Problem-solving Method Development Language*) [104, 196], provee las bases para la descripción de los componentes de conocimiento para así facilitar el diseño, la composición y la reutilización de WFs. El segundo metamodelo, la ontología de organización, que forma parte del TOVE Ontology Project[1], da soporte a la estructuración y organización de los participantes en la ejecución del WF. Finalmente, el tercer pilar del metamodelo da soporte a la estructuración de los procesos, es decir, a la ordenación de sus actividades. Esta última parte del modelo estará a cargo de las ontologías de redes de Petri descritas en el capítulo 3.

4.1. Arquitectura del metamodelo

Ya en 1991 un grupo de investigadores americanos discutieron el estado de la tecnología de los SBCs [192]. De su análisis concluyeron que dado el incremento de la complejidad de las aplicaciones basadas en conocimiento, era de crucial importancia orientar el desarrollo de los sistemas basados en conocimiento (SBCs) hacia un modelo centrado en la reutilización. En la opinión de estos investigadores, a finales de los años ochenta la tecnología de los SBCs había alcanzado un punto donde el coste de construir una aplicación desde cero era demasiado importante y era necesario buscar alternativas tecnológicas que permitiesen reducir el coste de desarrollo. La principal alternativa para conseguir este objetivo fue orientar el desarrollo de los nuevos sistemas hacia la reutilización y la compartición de conocimiento. Al construir una aplicación a partir de componentes robustos y preexistentes, no sólo se conseguiría reducir el coste de desarrollo sino también el de evaluación y mantenimiento. Desde entonces los marcos conceptuales han evolucionado para hacer de la reutilización el

[1] http://www.eil.utoronto.ca/enterprise-modelling/tove

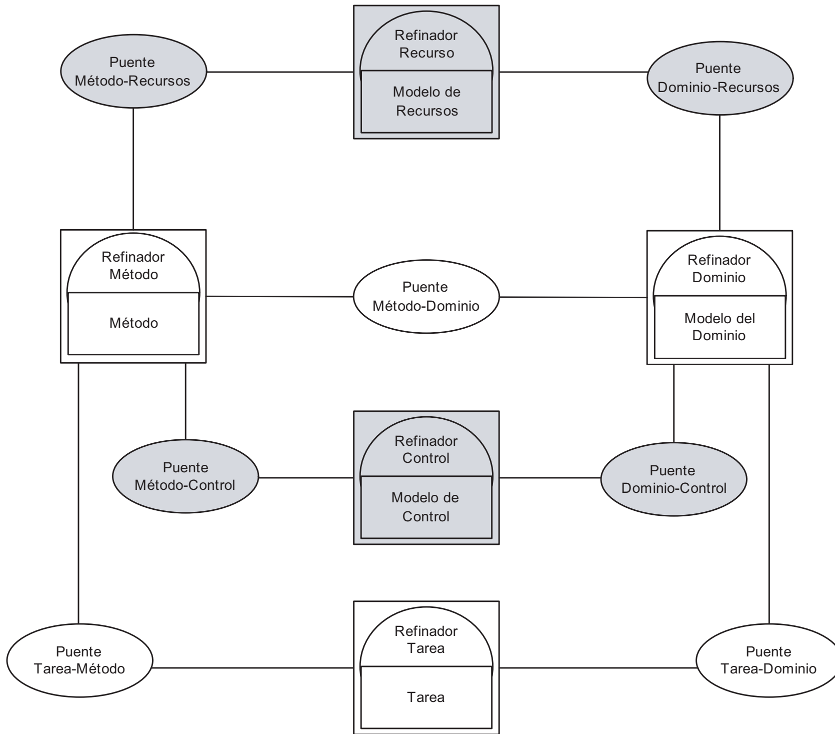


Figura 4.1: Marco de conocimiento para la descripción de WFs. Los elementos coloreados en gris han sido añadidos a la arquitectura definida en UPML.

punto de partida del desarrollo de SBCs. Prueba de ello son los marcos conceptuales definidos en [113, 196, 228] que proveen metamodelos y librerías para facilitar la construcción de los SBCs.

El metamodelo que se presenta a continuación sigue esta filosofía y orienta la construcción de WFs a partir de los componentes reutilizables en el nivel de conocimiento. La Figura 4.1 muestra el metamodelo propuesto, que adapta la aproximación de UPML [196] al modelado y ejecución de WFs. La figura sólo representa los pilares o los elementos más significativos del metamodelo, el cual contiene muchos más conceptos que se irán introduciendo a lo largo de este capítulo.

4.1.1. Componentes de conocimiento

En la arquitectura representada en la Figura 4.1 cada una de las cajas identifica un componente de conocimiento reutilizable, de forma que distintos WFs pueden usar los mismos componentes en su definición. Cada uno de estos componentes captura un aspecto esencial del modelado de un WF:

- *Tareas.* Identifican el problema a resolver y son descripciones de alto nivel de las características un WF como sus entradas, salidas y condiciones de aplicabilidad.
- *Métodos.* Son descripciones sobre la forma de resolver un determinado problema (o tarea). En función de su complejidad el método puede descomponer el problema en (sub)tareas más sencillas.
- *Modelo de control.* Es una descripción de una estructura de control que permite coordinar la ejecución de un conjunto de tareas. Este modelo se especifica a través de redes de Petri para que los patrones más relevantes de los WFs puedan representarse adecuadamente dentro del metamodelo.
- *Modelo del dominio.* Permite especificar los hechos, las relaciones, reglas y asunciones que se aplican en el dominio de la aplicación.
- *Modelo de recursos.* Describe a los agentes (software y humanos) que participan en la ejecución del proceso y que son los responsables de realizar tareas cuya resolución no se descompone en (sub)tareas. Estos agentes se clasifican en función de las unidades organizativas a las que están asignados y de los papeles que juegan en la ejecución del WF.

Los cinco componentes anteriores extienden el concepto `KnowledgeComponent` que describe, siguiendo la idea central de este modelo, a un elemento de conocimiento genérico y reutilizable:

```
subClassOf(KnowledgeComponent, Concept).

objectProperty(pragmatics).
domain(pragmatics, KnowledgeComponent).
range(pragmatics, Pragmatics).
cardinality(pragmatics, 1).

objectProperty(ontologies).
domain(ontologies, KnowledgeComponent).
range(ontologies, Ontology).
minCardinality(pragmatics, 1).
```

Cada componente de conocimiento utiliza el campo `pragmatics` para describir sus metadatos siguiendo las recomendaciones de la iniciativa *Dublin Core Metadata*[2]. A través de esta relación se identificarán el nombre del componente, su descripción, sus objetivos, sus autores y más información de soporte. Además, todos los componentes de conocimiento se describen a través de al menos una ontología.

4.1.2. Uso de ontologías en el metamodelo

Las ontologías [122, 121] son un tipo especial de componente de conocimiento que no está incluido en la Figura 4.1 pero que juega un papel fundamental en la capacidad

[2]http://purl.org/dc/

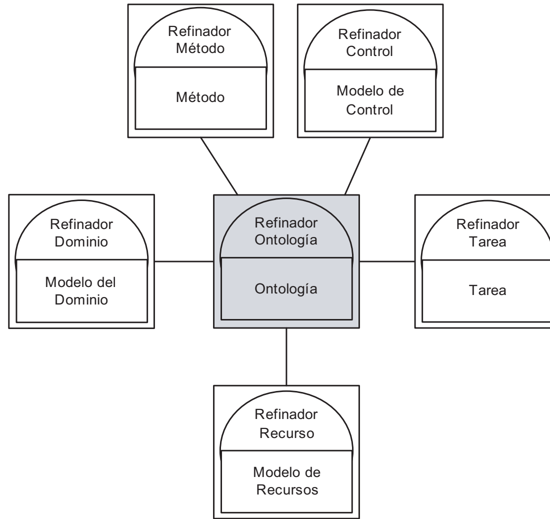


Figura 4.2: El papel de las ontologías en la arquitectura del metamodelo consiste en describir formalmente al resto de los componentes de conocimiento

de reutilización del metamodelo. Podría decirse que las ontologías son el núcleo del metamodelo tal como se puede apreciar en la Figura 4.2, dado que proporcionan la terminología utilizada para describir las características de los componentes de conocimiento. Así, cada componente estará descrito mediante una o varias ontologías, con lo que sus características estarán especificadas formalmente y serán interpretables por razonadores lógicos. Las ontologías utilizadas para describir un determinado componente de conocimiento estarán incluidas en la relación `ontologies`:

```
objectProperty(ontologies).
domain(ontologies, KnowledgeComponent).
range(ontologies, Ontology).
```

Una ontología permite especificar un conjunto de conceptos, relaciones y axiomas acerca de un determinado dominio. Por ejemplo, la Figura 4.3 muestra parte de una ontología de materiales, en la que se especifican conceptos como *Material*, *Madera* o *Aglomerado*, y relaciones como *composición* o *resistencia*. Además, también se incluyen individuos como *nogal*, que pertenece a la clase *Madera*, y axiomas para restringir los valores que pueden tomar las relaciones. Al especificar la terminología de forma independiente a su uso, esta ontología podría usarse para describir una tarea de gestión del almacén o una tarea de fabricación. La clase `Ontology` hace referencia al concepto de ontología:

```
subClassOf(Ontology, KnowledgeComponent).

objectProperty(signature).
```

Ontology materiales
pragmatics
Materiales empleados en el dominio de la fabricación de muebles a medida.
signature elements
classes
Material, Madera, MaderaBlanda, MaderaDura, MaderaConifera, MaderasFrondosas, MaderasExóticas, Aglomerado, Conglomerado, Fibra, MDF, Metal, MetalLigero, MetalPesado, Vidrio, Laca, Cola, PolimeroNatural, PolímeroSinteticos, ...
object properties
composición, impurezas, relleno, contrachapado, laminado, alveolado, acabado, secado, pintura, revestimiento, tratamiento por inmersión, tratamiento por inyección, aleación, barnizado, vetado, ...
data properties
resistencia, peso, resistenciaPeso, elasticidad, fatiga, tensión, compresionAxil, traccionAxil, compresionTransversal, traccionTransversal, flexionEstatica, humedad, durabilidad, deformabilidad, dilatación, resistenciaElectrica, resistenciaAgua, resistenciaFuego, ...
individuos
pinoNegral, pinoTea, pinoSilvestre, sabina, tejo, abeto, roble, haya, olmo, chopo, haya sauce, castaño, nogal, aliso, ébano, wenge, okume, ukola, caoba, balsa, mdf, acero, aluminio, cobre, cromo, hierro, bronce, plata, oro, estaño, ...
...
axioms
$\forall m, n\ maderaConifera(m) \wedge categoria(m, 1) \wedge compresionAxil(m, n) \rightarrow n < 130$
$\forall m, n\ maderaConifera(m) \wedge categoria(m, 1) \wedge compresionTransversal(m, n) \rightarrow n < 25$
$\forall m, n\ maderaConifera(m) \wedge categoria(m, 2) \wedge compresionAxil(m, n) \rightarrow n < 105$
$\forall m, n\ maderaConifera(m) \wedge categoria(m, 2) \wedge compresionTransversal(m, n) \rightarrow n < 20$
$\forall m, n\ maderaFrondosa(m) \wedge categoria(m, 1) \wedge compresionAxil(m, n) \rightarrow n < 135$
$\forall m, n\ maderaFrondosa(m) \wedge categoria(m, 1) \wedge compresionTransversal(m, n) \rightarrow n < 50$
$\forall m, n\ maderaFrondosa(m) \wedge categoria(m, 2) \wedge compresionAxil(m, n) \rightarrow n < 110$
$\forall m, n\ maderaFrondosa(m) \wedge categoria(m, 2) \wedge compresionTransversal(m, n) \rightarrow n < 45$
...

Figura 4.3: Ejemplo de una ontología de materiales

```
domain(signature, Ontology).
range(signature, Signature).
```

La relación `signature` asociada a la ontología representa su firma, la cual dependerá del formalismo elegido para representarla. En nuestro caso, se utilizará la lógica de predicados para definir esta firma, de forma que una ontología estará compuesta por un conjunto de hechos y de axiomas que se describen a través de sus correspondientes predicados y reglas. La clase `Signature` permite aglutinar estos hechos y axiomas:

```
subClassOf(Signature, Concept).

objectProperty(signatureElements).
domain(signatureElements, Signature).
range(signatureElements, SignatureElement).

union(SignatureElement, [Class, Individual, ...]).
```

```
objectProperty(axioms).
domain(axioms, Ontology).
range(axioms, Formula).
```

Para estructurar el conocimiento los hechos están clasificados en los tipos de constructores identificados en OWL (que están siendo usados para definir el metamodelo), de modo que una firma estará compuesta por un conjunto de elementos que representan clases, relaciones, individuos, etc. (relación `signatureElements`). Es importante mencionar que la semántica asociada a estos hechos estará a cargo del motor de inferencia/razonador, ya que nuestro metamodelo no está ligado a ningún lenguaje de ontologías (como OWL [84], WSML [47]). Por ejemplo, el hecho $subClassOf(A, B)$ indica que A es subclase de B, y que esta interpretación estará a cargo del razonador que dé soporte al metamodelo. En el caso de que este razonador fuese FLORA-2, se haría a través de la siguiente regla: `X::Y :- subClassOf(X,Y)`, donde '::' indica la relación *es-un* entre la clase B y la A.

La relación `axioms` se refiere a las restricciones o axiomas que se aplican a los conceptos y relaciones de la ontología, y su interpretación también estará a cargo del razonador. Por ejemplo, el razonador ha de dotar al axioma $\forall m, n\ maderaConifera(m) \wedge categoria(m, 1) \wedge compresionAxil(m, n) \rightarrow n < 130$ de una interpretación en el nivel de representación del conocimiento. Esta interpretación puede ser directa si el lenguaje del razonador soporta axiomas, pero por norma general, las capacidades de cada razonador son distintas y por ello la interpretación de estos axiomas o fórmulas puede ser compleja. Por ejemplo, el razonador FLORA-2 no soporta la definición de axiomas, de forma que su representación ha de hacerse a través de reglas que permitan identificar el error producido al no verificarse la restricción. Así, para el axioma anterior sería necesario definir la regla `error(ID1,X,Y) :- maderaConifera(X), categoria(X,1), compresionAxil(X,Y), not Y<130`.

4.1.3. Ontologías que soportan la arquitectura del metamodelo

La Figura 4.4 representa la pila de ontologías sobre la que está construido el metamodelo. El primer nivel especifica los tipos de datos primitivos y está basada en la especificación *XML Schema Data Types* [33], que contiene tipos como los números enteros, números reales, caracteres, cadenas de texto o fechas. El segundo nivel está construido sobre la ontología de tipos de datos y provee un modelo de representación de conocimiento basado en un determinado formalismo lógico. Por ejemplo, los modelos de OWL o WSML están basados en lógicas descriptivas. Los demás elementos de la pila de ontologías se utilizan para definir los componentes de conocimiento y los adaptadores propuestos en este metamodelo:

- Ontología *UPML* [196]. Define las tareas, métodos, modelos del dominio, ontologías y adaptadores. La razón por la cual se ha optado por el uso de ontologías para dar cobertura a la arquitectura del metamodelo se debe a que UPML está

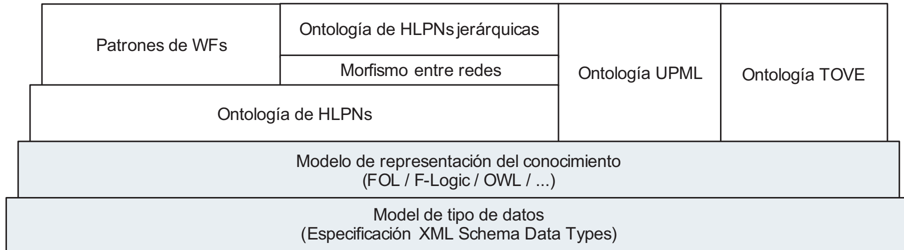


Figura 4.4: Pila de ontologías que describen la semántica del metamodelo

descrito a través de una ontología, que, como se ha comentado anteriormente, es el núcleo de la aproximación.

- Ontología *TOVE*[3] [109]. Permite clasificar y especificar/caracterizar a los agentes que participan en la ejecución de un WF. Esta ontología es lo suficientemente genérica como para representar los recursos de muchos dominios de aplicación, de modo que si no se pueden describir algunos aspectos se podrían especificar los conceptos de esta ontología para dar cobertura a la dimensión de recursos.
- Ontología de *HLPNs* [266]. Se usa para representar las redes de Petri utilizadas en la definición del modelo de control.
- Ontología de *Patrones de WFs*. Define los conceptos que modelan los patrones de WFs descritos en el Capítulo 2 y los axiomas que restringen la estructura y el comportamiento de las HLPNs que los implementan.
- Ontologías de *redes jerárquicas* y morfismo entre redes. Aporta los conceptos para definir HLPNs jerárquicas y consecuentemente definir el modelo de control. Debido a que las redes jerárquicas no tienen semántica operacional la capa *morfismo entre redes* permite interpretar una red jerárquica como una HLPN.

4.1.4. Adaptadores

Cada componente de conocimiento del metamodelo (es decir las tareas, los métodos, y los modelos de control, de recursos y del dominio) se puede describir independientemente de los demás componentes a través de *adaptadores*, que permiten relacionar las distintas partes de la especificación entre sí. Se podría decir que los adaptadores son los responsables de combinar de forma consistente las distintas dimensiones de un WF para así adaptar su comportamiento al problema a resolver. La clase `Adapter` modela este componente:

```
subClassOf(Adapter, BinaryRelation).
```

[3] http://www.eil.utoronto.ca/enterprise-modelling/tove

```
domain(pragmatics, Adapter).
domain(ontologies, Adapter).
```

Los adaptadores son relaciones binarias que se describen, al igual que los demás componentes de la arquitectura, a través de las relaciones `pragmatics` y `ontologies`. La adaptación tiene lugar entre dos componentes de conocimiento, de ahí que sea una relación binaria, a los que se hace referencia a través de `argument1` y `argument`:

```
objectProperty(argument1).
domain(argument1, Adapter).
range(argument1, KnowledgeComponent).

objectProperty(argument2).
domain(argument2, Adapter).
range(argument2, KnowledgeComponent).
```

Es conveniente comentar que una buena parte de la adaptación es terminológica: los componentes de conocimiento se describen mediante ontologías que pueden ser diferentes, y por ello es necesario que exista un mecanismo que permita establecer relaciones de equivalencia entre esas ontologías. La relación `renamings` capturará las discrepancias terminológicas entre los componentes de conocimiento:

```
objectProperty(renaming).
domain(renaming, Adapter).
range(renaming, Renaming).
```

Las instancias de la clase `Renaming` son las encargadas de establecer este tipo de correspondencias, que son de tipo 1:1, es decir, término a término:

```
subClassOf(Renaming, Concept).

objectProperty(in).
domain(in, Renaming).
range(in, SignatureElement).
cardinality(in, 1).

objectProperty(out).
domain(out, Renaming).
range(out, SignatureElement).
cardinality(out, 1).
```

Estas correspondencias son de gran utilidad para tratar conceptos con el mismo significado, pero que están definidos en contextos o dominios diferentes. Por ejemplo, un componente de conocimiento puede utilizar el término *comprador* y otro el término *persona* cuando en realidad ambos se están refiriendo a un *usuario*. A través de una instancia de la clase `Renaming` un adaptador podría indicar que los dos términos anteriores significan lo mismo:

```
Renaming(usuario).
in(usuario, comprador).
out(usuario, persona).
```

El metamodelo proporciona dos tipos de adaptadores: *puentes* y *refinadores*. Los puentes (*bridge*, en inglés) se usan para unir/relacionar componentes de conocimiento de *distinto* tipo. La Figura 4.1 representa los siete puentes de la arquitectura mediante elipses:

- *Puente Tarea-Dominio*. Indica que la tarea se puede aplicar al modelo del dominio y su principal función es adaptar la terminología de la tarea a la del dominio de aplicación. Es necesario matizar que esta adaptación no tiene por qué ser sencilla, ya que puede referirse a fórmulas que afectan a las condiciones y asunciones de la tarea.

- *Puente Método-Tarea*. Indica que método o métodos permiten resolver una tarea. En este sentido, un método puede solucionar varias tareas y una tarea puede ser resuelta por distintos métodos. Por ejemplo, una tarea de planificación se puede resolver a través de varios métodos de búsqueda. Para cada uno de los métodos será necesario establecer las correspondencias entre las entradas y salidas de la tarea y del método.

- *Puente Método-Dominio*. Establece las relaciones entre las terminologías del método y del dominio. Dado que la mayor parte de la terminología utilizada por el método ya está asociada transitivamente con la del dominio a través de los puentes que relacionan *(i)* el método con la tarea que resuelve y *(ii)* la tarea con el dominio en el que se aplica, la principal función de este puente es establecer nuevas asunciones que limiten el espacio de búsqueda del proceso de resolución del problema. Por ejemplo, asociando las entradas del método con determinados hechos o relaciones del modelo del dominio.

- *Puente Método-Control*. Indica que un modelo de control representa la descripción operacional de un método. Así, un mismo modelo de control (red de Petri) podrá utilizarse para especificar la ejecución de distintos métodos. La principal función de este puente es establecer las correspondencias entre la red de Petri y la terminología del método. Por ejemplo, asociando las precondiciones del método con la condición de guardia de alguna transición del modelo de control.

- *Puente Método-Recursos*. Este puente, además de indicar las correspondencias terminológicas entre el método y el modelo de recursos, establece las características que deben cumplir los agentes para participar en la ejecución de un método. Por ejemplo, permite indicar que para la ejecución de un determinado método primitivo es necesario que el agente tenga ciertas habilidades y un determinado papel.

- *Puente Dominio-Control*. Indica las relaciones entre la terminología usada para describir el modelo de control y la del modelo del dominio. Estas correspondencias

cias permiten establecer el álgebra de la red de control a partir del conjunto de términos de las ontologías del modelo del dominio.

- *Puente Dominio-Recursos.* Indica las relaciones entre la terminología usada para describir el recurso y la del modelo del dominio. La principal función de este puente es relacionar las habilidades asociadas a los agentes en el modelo de recursos con sus correspondencias en el modelo del dominio.

Por una parte, si los dos componentes de conocimiento son del mismo tipo, el adaptador se denomina *refinador* (*refiner*, en inglés), y se representa en la arquitectura del metamodelo mediante semicírculos acoplados al componente sobre el que se aplican. Cada uno de los cinco componentes de conocimiento tiene asociado un refinador, es decir, existen refinadores de tarea, de método, del modelo de control, del modelo del dominio y del modelo de recursos. Con este tipo de adaptador la definición de un nuevo componente no se realiza desde cero o copiando al original, sino que se especializa un componente de conocimiento aplicando un conjunto de cambios o restricciones sobre sus elementos. De esta forma, la adaptación y creación de componentes es más sencilla.

4.1.5. Dimensiones de la arquitectura

Los cinco componentes de conocimiento del metamodelo se pueden agrupar en tres bloques, tal y como está representado por los correspondientes círculos de la Figura 4.5 que representan cada una de las dimensiones de los WFs vistas en el Capítulo 1:

- *Dimensión de procesos.* Permite modelar el proceso, es decir, la estructura de control y composición del WF. Esta dimensión está compuesta por tres componentes, las *tareas*, los *métodos* y los *modelos de control*, y por los puentes entre ellos.

- *Dimensión de conocimiento del dominio.* Esta dimensión se representa mediante el *modelo del dominio* y permite especificar los hechos, las relacionees, reglas y asunciones que se aplican en el dominio de la aplicación.

- *Dimensión de recursos.* Permite definir y clasificar los agentes que van a participar en la ejecución de un determinado WF, y se representa mediante el *modelo de recursos.*

4.2. Dimensión de procesos

La noción de tarea es crucial para el modelado de conocimiento, ya que a la postre los SBCs se caracterizan y evalúan en función de las tareas que van a realizar. Por este motivo, la dimensión de procesos se ha creado en torno al concepto de *tarea genérica* desarrollado por Chandrasekaran [59]. Una tarea especifica un objetivo a resolver,

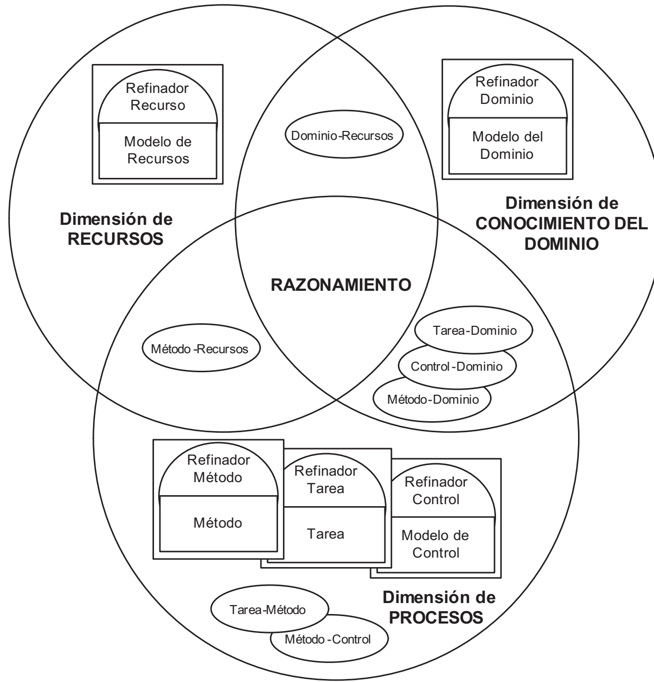


Figura 4.5: Relación entre los componentes de conocimiento y las dimensiones de los WFs

como la planificación del horario de clase o el diagnóstico de una enfermedad. El término *tarea genérica* se utiliza para referirse a la tarea especificada en el nivel de conocimiento y que es independiente del dominio de la aplicación. El concepto de tarea inicialmente propuesto también podía incluir lo que se denominaba *cuerpo de la tarea*, que describía de forma paramétrica el método usado para conseguir el objetivo de la tarea. Esta descripción dio lugar a la taxonomía representada en la Figura 4.6, donde las tareas que contienen la solución al problema pasaron a llamarse *tareas ejecutables*.

La aproximación de UPML, que es en la que está basada la dimensión de procesos, va más allá y separa explícitamente los métodos de su descripción operacional añadiendo otro nivel de reutilización. De esta forma, esta dimensión aproxima la descripción de los WFs en tres niveles de abstracción, tal y como se muestra en la Figura 4.7, donde además se representan los puentes que permiten relacionar los niveles adyacentes entre sí. De este modo:

- Las tareas constituyen el nivel de mayor abstracción y describen el problema que resuelve el WF en términos de sus entradas, salidas y condiciones. Por ejemplo, si un director de escuela quiere crear el horario de clases del curso, buscará una tarea que resuelva su problema de planificación dentro de una librería de componentes. Esta tarea, sin embargo, no le indicará la forma de resolver el

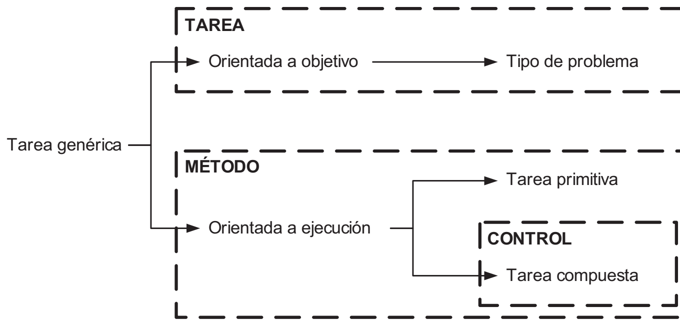


Figura 4.6: Taxonomía de tarea genérica definida por Chandrasekaran [59]. La taxonomía está anotada con rectángulos de forma que se puedan identificar las correspondencias entre la tarea genérica y los elementos definidos en metamodelo

problema. Para ello, tendrá que seleccionar un determinado puente entre el nivel de método y el de tarea. De forma simplificada, un puente *método-tarea* asocia un método a una tarea y define las correspondencias entre los términos que anotan las entradas, salidas y condiciones de ambos componentes.

- Los métodos constituyen el nivel intermedio de abstracción, y aportan la solución a un problema, o lo que es lo mismo, a una tarea. Si esa solución es compleja, los métodos describen la descomposición del problema, que al contrario de lo que se podría pensar, se especifica a través de (sub)tareas y no de (sub)métodos. Así, un método compuesto detalla realmente una solución parcial, ya que no indica cómo se resuelven las (sub)tareas que lo componen. Por ejemplo, la Figura 4.7 muestra un método compuesto que se descompone en tres (sub)tareas pero no presenta la forma de resolverlas: la solución no será completa hasta que no se asocie un método no compuesto (a través de los correspondientes puentes) a cada una de las (sub)tareas que participan en la resolución del problema.

- Finalmente, la coordinación de la resolución de las tareas que componen un método constituye el nivel más bajo de abstracción. Esta coordinación se consigue a través de a un conjunto de HLPNs agrupadas en una HLPN jerárquica. Como se comentó en el Capítulo 3, este tipo de redes son una poderosa herramienta para la descripción de sistemas complejos, ya que además de ser un formalismo matemático, también tienen una representación gráfica. Así, los diseñadores de un WF pueden establecer la coordinación entre las tareas a partir de los nodos, los arcos y la anotación de las redes. La integración entre el control y el método se consigue a través de un nuevo puente encargado de asociar las entradas, salidas y condiciones de los métodos con los elementos del grafo y las anotaciones de la HLPN.

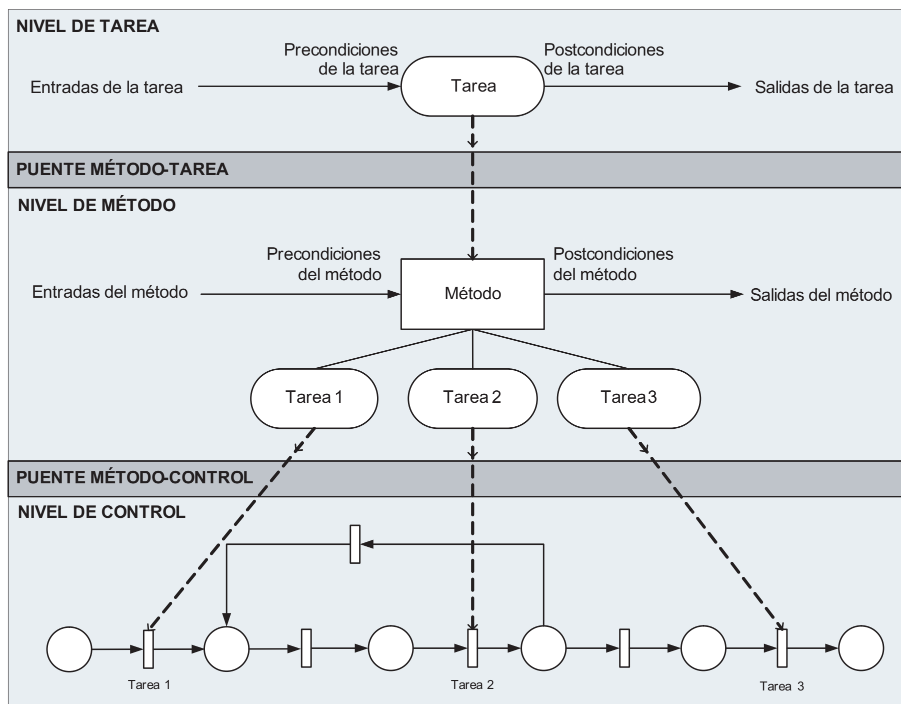


Figura 4.7: Niveles de abstracción que intervienen en la definición de un WF

4.2.1. Tareas

Una tarea detalla el tipo de problema que se quiere resolver mediante la ejecución de un WF. Es, por lo tanto, una descripción funcional de las características del problema expresadas independientemente del dominio de la aplicación. Tal y como se muestra en la Figura 4.8, las tareas suelen identificarse con clases genéricas de problemas, *grandes y complejos*, que pueden ser resueltos por métodos. Es necesario mencionar que en el dominio de los WFs existen más clases que las identificadas en la figura anterior, ya que también recogen problemas más sencillos como pueden ser los típicos de cualquier actividad administrativa.

El concepto `Task` da soporte a la definición de las tareas, las cuales se corresponden con las tareas orientadas a un objetivo definidas por Chandrasekaran:

```
subClassOf(Task, KnowledgeComponent).

objectProperty(inputRoles).
domain(inputRoles, Task).
range(inputRoles, Individual).

objectProperty(outputRoles).
domain(outputRoles, Task).
```

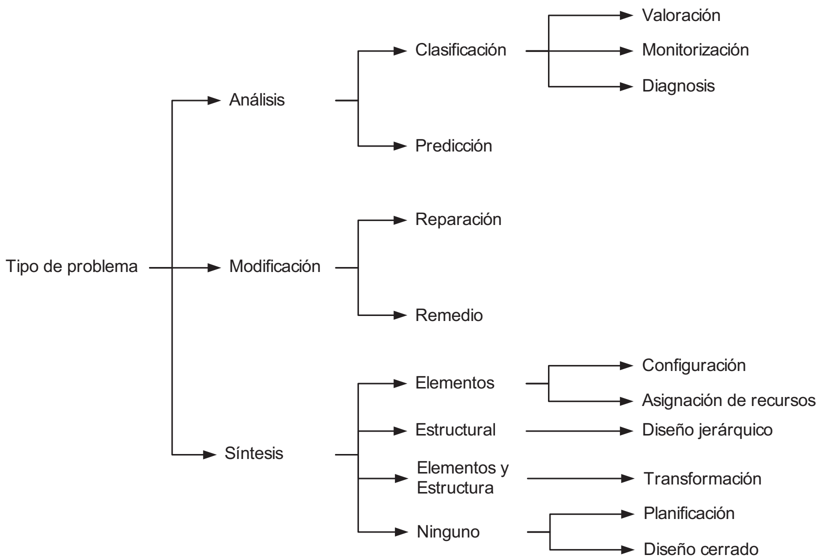


Figura 4.8: Taxonomía de tipos de problemas definida por la metodología KADS-1 [1, 228]

```
range(outputRoles, Individual).
```

La funcionalidad de una tarea se describe mediante sus entradas, salidas, condiciones de aplicabilidad y su objetivo. Las relaciones `inputRoles` y `outputRoles` hacen referencia a las entradas y salidas de la tarea respectivamente. Por ejemplo, la tarea representada en la Figura 4.9 tiene dos roles de entrada, denominados *trabajos* y *recursos*, y un rol de salida, denominado *plan*. La definición de los elementos está simplificada en la figura para facilitar la comprensión del ejemplo. Así, en este caso la especificación completa del rol de entrada *recursos* de la Figura 4.9 sería la siguiente:

```
Individual(recursos).
inputRoles(planificar, recursos).
```

donde la primera sentencia indica que la constante *recursos* es un individuo y la segunda asocia dicha constante al rol de entrada de la tarea *planificar*. La definición de los individuos tal y como se aprecia en el ejemplo anterior, también estará oculta en las demás descripciones de componentes de conocimiento que se verán a lo largo de este capítulo.

Las tareas especifican sus condiciones de aplicabilidad a través de la relación `competence`:

```
objectProperty(competence).
domain(competence, Task).
range(competence, Competence).
```

Esta relación trata de fijar *(i)* las condiciones que las entradas de la tarea deben cumplir para que su ejecución sea correcta (precondición), y *(ii)* las condiciones que las salidas cumplirán una vez la tarea haya sido resuelta (postcondición). Estas condiciones se establecerán a través de una instancia de la clase `Competence`:

```
subClassOf(Competence, Concept).

objectProperty(preconditions).
domain(preconditions, Task).
range(preconditions, Formula).

objectProperty(postconditions).
domain(postconditions, Task).
range(postconditions, Formula).
```

Las relaciones `preconditions` y `postconditions` se especifican a través de fórmulas en las que únicamente se puede hacer referencia a las condiciones que deben cumplir los roles de entrada y de salida de la tarea. Por ejemplo, las dos precondiciones de la tarea *planificar* indican que el conjunto de recursos y de trabajos no pueden estar vacíos. Estas fórmulas pueden ser más complejas: ejemplo de ello es la postcondición de esa misma tarea, que indica que las programaciones (*schedule*, en inglés) del plan se construyen combinando los recursos y los trabajos de la entrada.

Las tareas y sus objetivos están muy ligados al concepto de asunción, que permite especificar el conocimiento *asumido* (que no es necesario demostrar) sobre la tarea y que suele referirse a sus objetivos. A través de la relación `assumptions` se asociarán las fórmulas que contienen dicho conocimiento:

```
objectProperty(assumptions).
domain(assumptions, Task).
range(assumptions, Formula).
```

Por ejemplo, la tarea representada en la Figura 4.9 tiene por objetivo planificar a partir de los trabajos y recursos disponibles. Para que este objetivo se cumpla se define una asunción que indica que la planificación se resolverá si y sólo si los conjuntos de trabajos y recursos no son vacíos.

4.2.2. Métodos

En las primeras aproximaciones al modelado del conocimiento [59, 296] la noción de tarea se usó para referirse tanto a la funcionalidad como a la operacionalidad para resolver un problema. Tal y como se ha comentado anteriormente, se hizo evidente que separando ambos aspectos era posible desarrollar marcos conceptuales con los que se mejoraba la reutilización de los componentes de conocimiento [240, 60, 26]. A la parte encargada de describir la operacionalidad de una tarea se la denominó *método*. Una definición más precisa de método [26] diría que:

Task planificar
pragmatics
Planificar la asignación de recursos a un conjunto de trabajos.
ontologies
planificacion
input roles
trabajos, recursos
output roles
plan
preconditions
La entrada *trabajos* no puede ser un conjunto vacío:
$\exists e\, miembroDe(trabajos, e)$
La entrada *recursos* no puede ser un conjunto vacío:
$\exists e\, miembroDe(recursos, e)$
postconditions
Cada programación del plan asocia un recurso con un trabajo a realizar:

$$\forall s, r, t\ tieneProgramacion(plan, s) \wedge recursoEncargado(s, r) \wedge trabajoRealizado(s, t) \rightarrow miembroDe(recursos, r) \wedge miembroDe(trabajos, t)$$

assumptions
Siempre se puede planificar si los conjuntos de recursos y trabajos no son vacíos:

$$\forall o\ planificar(trabajos, recursos, o) \leftrightarrow \exists m_1, m_2\, miembroDe(trabajos, m_1) \wedge miembroDe(recursos, m_2)$$

Figura 4.9: Ejemplo de tarea de asignación de recursos

> *... especifica, independientemente del dominio y en el nivel de conocimiento, el comportamiento que resuelve una determinada clase de problemas.*

Esta definición enfatiza un aspecto importante de los métodos: un método describe el conocimiento relativo a la forma en la que se resuelve un problema y no tiene en cuenta una implementación. En este punto, uno podría preguntarse ¿qué describe ese conocimiento dinámico y por qué separarlo de la implementación? La respuesta a esta pregunta depende de si el método es primitivo o compuesto.

En los métodos compuestos el conocimiento está asociado a *(i)* la forma en la que se descomponen las tareas en (sub)tareas, y *(ii)* al control que coordina la ejecución de dichas (sub)tareas. La estructura de un método no varía en función del dominio o de la tarea en la que se está aplicando y además es independiente de la resolución de las (sub)tareas. La razón de separar este conocimiento de la implementación se debe a que con ella se incrementa la reutilización de los métodos. Por ejemplo, la Figura 4.10 muestra un método que resuelve la tarea de planificación representada en la Figura 4.9. Al describirse de forma independiente a la tarea y al dominio de la aplicación, este mismo método podrá ser reutilizado para resolver otros problemas similares en dominios parecidos como, por ejemplo, la planificación del horario de clases de un colegio o el reparto de una empresa de mensajería.

En los métodos primitivos el conocimiento dinámico se restringe a los roles de conocimiento y a las asunciones que debe verificar su implementación. Estos métodos no descomponen una tarea en otras (sub)tareas y, por lo tanto, cuando una tarea se resuelve a través de un método *primitivo* su ejecución se hace en un único paso. Ahora bien, la implementación del método que está asociada a esta ejecución se trata como un elemento externo al metamodelo (queda fuera del nivel de conocimiento [193]) y, como se verá en el Capítulo 6 estará asociada a los recursos del WF.

El concepto `Method` captura estas ideas y describe un método a partir de sus entradas, salidas y pre/postcondiciones:

```
subClassOf(Method, KnowledgeComponent).

domain(inputRoles, Method).
domain(outputRoles, Method).
domain(competence, Method).
```

4.2.2.1. Métodos compuestos

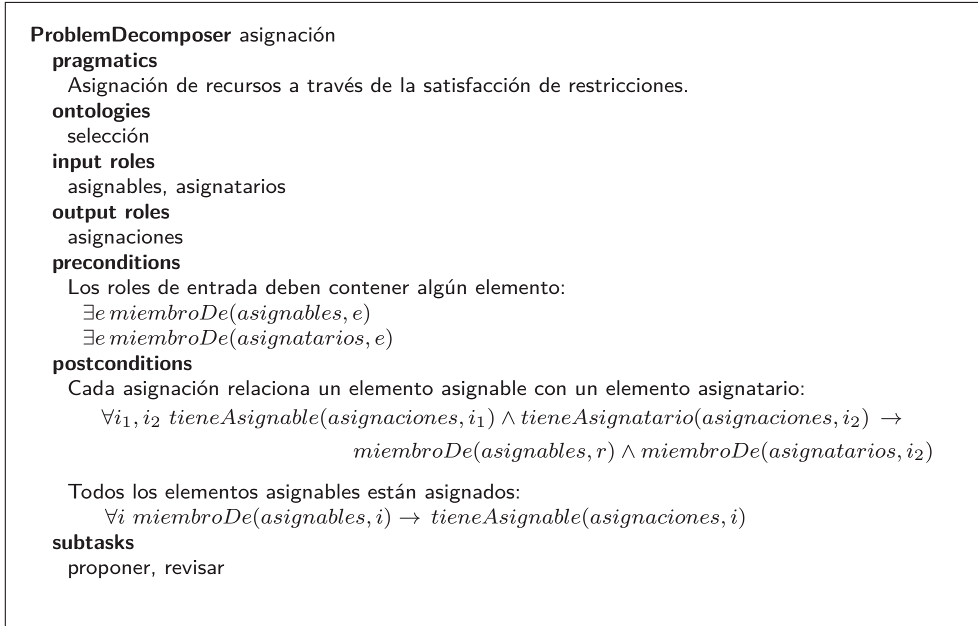
Un método compuesto no puede ejecutarse en un único paso de inferencia y por ello necesita descomponer la tarea que resuelve en (sub)tareas a través de las cuales consigue alcanzar la solución [104]. El concepto `ProblemDecomposer` modela este tipo de métodos y la relación `subTasks` hace referencia a las (sub)tareas en las que se descompone una tarea:

```
subClassOf(ProblemDecomposer, Method).

objectProperty(subTasks).
domain(subTasks, ProblemDecomposer).
range(subTasks, Task).
minCardinality(subTasks, 1).
```

Es importante resaltar nuevamente que la descomposición en (sub)tareas y no en (sub)métodos facilita en gran medida la reutilización, ya que independiza a los métodos de la forma en la que se va a resolver cada una de estas (sub)tareas. Por ejemplo, si un método tiene tres (sub)tareas y cada una de ellas se puede resolver a través de otros dos métodos distintos, existirán $2! \cdot 2! \cdot 2!$ configuraciones diferentes para resolver la tarea general asociada al método compuesto. En cualquier caso, es necesario precisar que aunque existan 8 configuraciones distintas su aplicabilidad dependerá de si cumplen todas las precondiciones y asunciones de la tarea. Además, esta estructuración facilita la integración de nuevos métodos en el sistema: por ejemplo, al añadir un método que soluciona una determinada tarea, también se están añadiendo nuevas configuraciones aplicables a métodos en cuya descomposición se encuentra dicha tarea.

Uno de los principales cambios motivados por la adaptación de UPML al modelado de WFs es la necesidad de separar explícitamente el control (descripción operacional)



ProblemDecomposer asignación
pragmatics
Asignación de recursos a través de la satisfacción de restricciones.
ontologies
selección
input roles
asignables, asignatarios
output roles
asignaciones
preconditions
Los roles de entrada deben contener algún elemento:
$\exists e\, miembroDe(asignables, e)$
$\exists e\, miembroDe(asignatarios, e)$
postconditions
Cada asignación relaciona un elemento asignable con un elemento asignatario:

$$\forall i_1, i_2\ tieneAsignable(asignaciones, i_1) \wedge tieneAsignatario(asignaciones, i_2) \rightarrow miembroDe(asignables, r) \wedge miembroDe(asignatarios, i_2)$$

Todos los elementos asignables están asignados:
$\forall i\ miembroDe(asignables, i) \rightarrow tieneAsignable(asignaciones, i)$
subtasks
proponer, revisar

Figura 4.10: Ejemplo de método compuesto para la asignación de recursos

de la competencia y descomposición de los métodos. Por ejemplo, el método compuesto representado en la Figura 4.10 no describe la descripción operacional: únicamente contiene las entradas, salidas, pre/postcondiciones, y (sub)tareas. En este caso, el ejemplo representa un problema de asignación que se resuelve a través de las tareas *proponer* y *revisar*.

En cambio, en UPML no existe una separación explícita entre la especificación del método y su descripción operacional. Los métodos compuestos tienen una relación denominada *operational description* a través de la cual especifican el control del método. Para UPML una descripción operacional se detalla a partir de un conjunto de roles intermedios y de programas. Sin embargo, en UPML no se profundiza en la definición del concepto *programa* y simplemente se indica que sirve para establecer el control entre las (sub)tareas. Esta indefinición ha motivado la aparición de distintas propuestas para describir este control como son los lenguajes CML2[16] o OCML[186], aunque a nuestro entender estas propuestas están más cerca del nivel de implementación que del de conocimiento. El siguiente ejemplo muestra en pseudocódigo la descripción operacional de un método compuesto que podría ser utilizado para proponer asignaciones:

```
ProblemDecomposer Proponer
  input roles: asignables, asignatario
  output roles: asignaciones
  subTasks: seleccionar-asignable, seleccionar-asignatario,
```

```
Operational description:
  mientras existan asignables no procesados
  entonces seleccionar-asignable
           repetir seleccionar-asignatario
           mientras asignatario ya haya sido asignado
           añadir asignación(asignable seleccionado, asignatario seleccionado)
                  a asignaciones
```

Uno de los problemas de la solución de UPML es que la descripción operacional pasa a ser un trozo de código *poco legible* por parte de personas ajenas al dominio informático. En este sentido, tal y como se comentó en el Capítulo 2, no se puede pedir a los diseñadores de WFs que conozcan la sintaxis de un lenguaje de programación. Otro de los problemas de UPML es que no separa la descripción operacional de la especificación del método lo cual reduce significativamente su capacidad de reutilización: aunque dos métodos tengan la misma estructura de control, no sería posible reutilizarla porque no existen puentes entre métodos y descripciones operacionales. En este punto, es necesario precisar que la capacidad de refinado de métodos en UPML es prácticamente nula, ya que no permite modificar la descomposición en tareas ni mucho menos cambiar el control [196]. Por todo ello, la separación de la descripción operacional de un método favorece enormemente la reutilización.

4.2.2.2. Métodos primitivos o simples

Este tipo de métodos se refiere a aquellos métodos que no descomponen tareas en (sub)tareas para alcanzar una solución, y se entienden como las primitivas de razonamiento o inferencia en el nivel de conocimiento. En UPML los métodos primitivos se modelan a través del concepto `ReasoningResource`, que añade dos nuevas relaciones a la descripción de un método:

```
subClassOf(ReasoningResource, Method).

objectProperty(knowledgeRoles).
domain(knowledgeRoles, ReasoningResource).
range(knowledgeRoles, Individual).

domain(assumptions, ReasoningResource).
```

La relación `knowledgeRoles` hace referencia a los roles de conocimiento. Este nombre puede llevar a equívoco, ya que usualmente suele utilizarse para referirse tanto a los roles dinámicos (entradas y salidas) como a los estáticos. En UPML, sin embargo, esta relación apunta exclusivamente a los roles estáticos, es decir, aquellos que son estables a lo largo del tiempo y que no suelen pasarse como parámetros. Por ello este conocimiento suele formar parte del propio método, de modo que cuando el método se asocia a un determinado dominio, estos roles se emparejarán con reglas u otro conocimiento estático del dominio de la aplicación. La Figura 4.11 muestra un método primitivo encargado de *seleccionar* un elemento de entre un conjunto de elementos elegibles. En este método se define un rol de conocimiento, denominado *criterios*, que identifica la estrategia de selección del elemento.

ReasoningResource seleccionar
pragmatics
Selección de un elemento de entre un conjunto de elementos seleccionables.
ontologies
selection
input roles
conjunto
output roles
objeto
knowledge roles
criterios
preconditions
El conjunto de entrada debe contener algún elemento:
$\exists e\, miembroDe(conjunto, e)$
postconditions
Se selecciona un objeto perteneciente al conjunto de objetos seleccionables:
$seleccionar(conjunto, objeto) \rightarrow miembroDe(conjunto, objeto)$
asunciones
Siempre existe un criterio de selección:
$\exists e\, miembroDe(criterios, e)$

Figura 4.11: Ejemplo de método primitivo de selección

La segunda relación, denominada `assumptions`, se utiliza para establecer el conocimiento asumido, y que por lo tanto, no es necesario demostrar. En el ejemplo de la Figura 4.11 la única fórmula asociada a las asunciones indica que siempre existe un objeto que cumple los criterios de selección. Aunque en muchas situaciones las asunciones puedan parecer innecesarias (en el ejemplo anterior exponen una condición bastante simple) sí tienen su importancia en el proceso de razonamiento, ya que reducen el espacio de búsqueda de las soluciones.

Es importante mencionar una última y muy importante característica de los métodos primitivos: *no* tienen implementación. Esto se debe a que la implementación no tiene interés desde la perspectiva del modelado de conocimiento a la que está orientada esta arquitectura. Sin embargo, también es cierto que una aplicación no puede funcionar sin tener asociada una implementación que permita ejecutar estas primitivas de razonamiento. Como se verá más adelante, el puente entre los métodos y los recursos se encargará de definir qué agente se hará cargo de la ejecución de los métodos primitivos.

4.2.3. Modelo de control

Este modelo permite representar estructuras de control mediante HLPNs jerárquicas que, a su vez, están modeladas a través de las ontologías de redes de Petri descritas en el Capítulo 3. Se trata, por lo tanto, de un modelo de control compuesto por un conjunto de HLPNs (planas) unidas a través de mecanismos de composición. Es

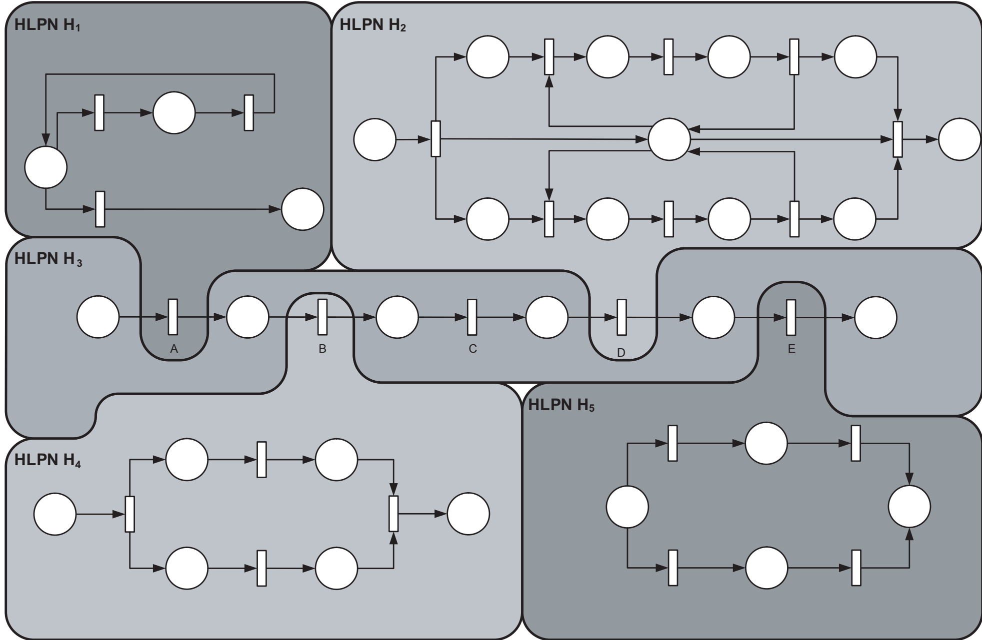


Figura 4.12: Ejemplo de diseño paramétrico usando HLPNs

importante resaltar que el modelo de control está especificado exclusivamente en términos de redes de Petri y que es *independiente* de las características de los métodos que puedan utilizarla como descripción operacional. En este sentido, la HLPN jerárquica es un componente de conocimiento que puede utilizarse para especificar el control operacional de diferentes métodos, a través de los correspondientes puentes entre ambos componentes.

En este apartado se esbozará únicamente la idea sobre la que se asienta el modelo de control. Dado que este modelo es la parte más compleja y novedosa del metamodelo de representación de WFs, es necesario prestar más atención a su descripción. Por este motivo, y para no romper el hilo argumental de este capítulo, el modelo de control se detallará en el Capítulo 5.

Al igual que los demás componentes de la arquitectura, el modelo de control (concepto `ControlModel`) se define como un componente de conocimiento pero también como una HLPN jerárquica, de modo que además de las relaciones `pragmatics` y `ontologies`, también hereda `hasPages`, `hasSubstitutions` y `hasFusions` de las redes jerárquicas:

```
subClassOf(ControlModel, KnowledgeComponent).
subClassOf(ControlModel, HHLPN).
```

```
ControlModel while2seq
  pragmatics
    WF con un bucle while y una secuencia de dos procesos.
  hasPages
    processPage, resultPage, whilePage, sequencePage
  hasSubstitutions
    runSubstitution, resultSubstitution, controlSubstitution
  hasFusions
    enabledPlaceFusion, finishedPlaceFusion, outputPlaceFusion, controlPlaceFusion
```

Figura 4.13: Ejemplo de modelo de control

Consecuentemente, un modelo de control estará compuesto por HLPN planas (*páginas*) unidas por mecanismos de composición (*sustituciones* y *fusiones*). El papel de las ontologías en este modelo es también importante: se utilizan como firmas de las páginas de la red jerárquica. Así, los elementos de las páginas (es decir, las plazas, las transiciones y los arcos) se anotarán mediante los hechos, y predicados de las ontologías.

Uno de los objetivos de este metamodelo es facilitar el diseño de WFs a los desarrolladores. En este sentido, aunque las redes de Petri son bastante intuitivas, es justo reconocer que su uso es complicado cuando las funcionalidades requeridas son complejas. Concretamente, las HLPNs que modelan algunos de los patrones de control avanzado de WFs no son sencillas ni intuitivas. Por este motivo, algunos lenguajes basados en redes de Petri, como YAWL [259], añaden una capa de abstracción para simplificar el uso de estos patrones. Si se combina esta última idea con el concepto de diseño paramétrico [187] en el que están basados los *métodos de resolución de problemas* (PSMs), la construcción de los WFs puede simplificarse. Así, la idea subyacente es que la creación del modelo de control es un proceso de ingeniería que consiste en unir piezas de control prefabricadas, de la misma forma en la que un circuito digital se monta a partir de puertas lógicas. Por ejemplo, la Figura 4.12 muestra gráficamente un ejemplo simplificado de modelo de control compuesto por cinco redes planas que se componen como si fuesen un puzzle en el que se encajan cada una de las piezas (en este caso cada una de las redes). Aunque esta red es un ejemplo simplificado, la construcción del modelo de control sigue la misma filosofía: un conjunto de piezas prefabricadas (que se corresponden con los patrones de control de WFs identificados en el Capítulo 2) se combinan entre si hasta formar un único puzzle (la descripción operacional de un método compuesto).

La Figura 4.13 muestra un ejemplo de modelo de control cuya representación gráfica está recogida en la Figura 4.14. En este caso el modelo está compuesto por cuatro HLPNs denominadas *processPage*, *resultPage*, *whilePage* y *sequencePage*, que están unidas por medio de tres sustituciones y cuatro fusiones.

La Figura 4.15 incluye la definición de las algunas de las páginas del modelo de

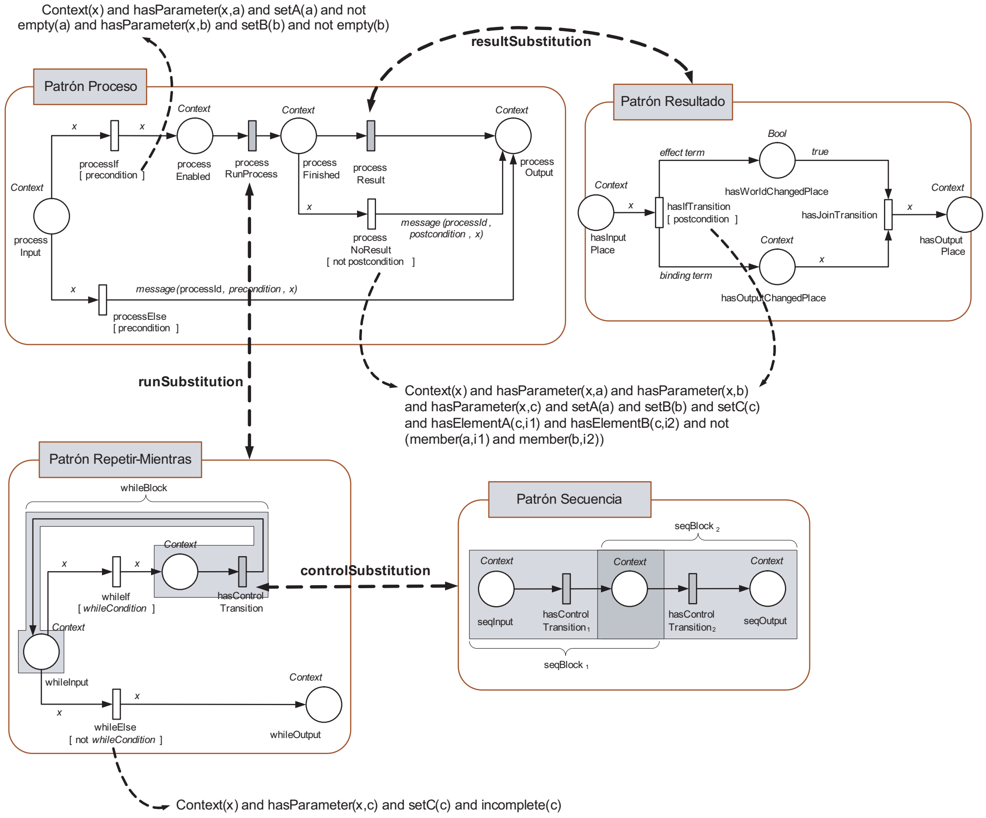


Figura 4.14: Representación gráfica del modelo de control de las figuras 4.13 y 4.15

control. En particular define las páginas *processPage*, *whilePage* y *sequencePage* para dar una idea de cómo se conceptualizan los patrones que componen el modelo de control y que se describen a través de un determinado concepto de la ontología. En el ejemplo anterior, la HLPN *processPage* es un individuo de la clase `ProcessPattern` que modela el patrón para la representación de un proceso. Asimismo, las HLPNs *whilePage* y *sequencePage* posibilitan la descripción de los patrones repetir-mientras y secuencia, respectivamente. Concretamente en este ejemplo se modela una secuencia de dos procesos capturados a través de los individuos *seqBlock1* y *seqBlock2*. La estructura de los patrones de control y su significado dentro del modelo se analizarán más detenidamente en el Capítulo 5.

Finalmente, cabe decir que esta parte del metamodelo es una de las principales aportaciones de esta tesis doctoral al modelo arquitectónico de los PSMs, y aunque en esta tesis se orienta al dominio de los WFs la solución es extrapolable al modelado de PSMs en la medida en que se integra perfectamente en la arquitectural UPML como un componente de conocimiento más.

```
ProcessPattern processPage
  hasInputPlace
  hasInputPlace
    processInput
  hasOutputPlace
    processOutput
  hasEnabledPlace
    processEnabled
  hasFinishedPlace
    processFinished
  hasIfTransition
    processIf
  hasElseTransition
    processElse
  hasRunProcessTransition
    processRun
  hasResultTransitions
    processResult
  hasNoResultTransition
    processNoResult
  ...

RepeatWhilePattern whilePage
  hasInputPlace

    whileInput
  hasOutputPlace
    whileOutput
  hasIfTransition
    whileIf
  hasElseTransition
    whileElse
  hasControlBlock
    whileBlock
  ...

SequencePattern sequencePage
  hasInputPlace
    seqInput
  hasOutputPlace
    seqOutput
  hasControlList
    seqBlock1, seqBlock2
  ...
```

Figura 4.15: Descripción parcial de algunas páginas del modelo de control de la Figura 4.13

4.3. Dimensión de conocimiento del dominio

Desde la perspectiva del modelado tradicional de WFs no se suele gestionar el conocimiento del dominio. Para la mayoría de los desarrolladores los WMSs son herramientas que permiten coordinar las actividades a ejecutar por parte de los participantes, pero que no tienen en consideración el conocimiento del dominio asociado al proceso de negocio. Por ello, estas herramientas no permiten manejar fácilmente el conocimiento del dominio, que suele integrarse (al igual que los demás datos) a través de los parámetros, variables y expresiones que anotan la estructura del WF. Si la estructura del conocimiento es compleja, esta integración es complicada e incluso en aquellos casos donde se consigue la reutilización de este conocimiento en otros WFs es poco probable al estar directamente codificado dentro de la estructura del WF.

En este apartado se describirá el componente que permite capturar este conocimiento dentro del metamodelo de representación de WFs propuesto. Al describirse independientemente de los demás componentes del metamodelo, el conocimiento del dominio podrá reutilizarse en distintos métodos, tareas y modelos de control.

4.3.1. Modelo del dominio

La descripción de un dominio de aplicación no consiste exclusivamente en hechos, sino que también puede contener conocimiento a modo de reglas, asunciones o axiomas. Para cubrir este aspecto en la descripción del dominio se hace uso de tres tipos de conocimiento:

- *Ontologías del dominio*. Especifican la terminología y la axiomática utilizada en la descripción de los conceptos del dominio. Por ejemplo, una ontología de materiales de madera contendrá la terminología que permita identificar los distintos tipos y substratos de madera, así como los axiomas que restrinjan los valores que pueden tomar cada uno de los conceptos y relaciones.
- *Conocimiento del dominio*. Especifica instancias/hechos del modelo del dominio que no están descritos en las ontologías. Por ejemplo, las máquinas y recursos de la planta de fabricación o su agenda de trabajo.
- *Metaconocimiento*. Especifica reglas, axiomas y asunciones que se dan en el dominio de aplicación. Por ejemplo, el conocimiento para determinar la ruta de fabricación de un mueble.

La clase `DomainModel` se refiere al conocimiento que permite modelar el dominio de la aplicación:

```
subClassOf(DomainModel, KnowledgeComponent).

objectProperty(knowledge).
domain(knowledge, DomainModel).
range(knowledge, Formula).
```

Al ser un componente de conocimiento hereda la relación `ontologies` que se utiliza para definir la terminología a aplicar. Los demás tipos de conocimiento se especifican a través de: hechos, relaciones y asunciones. Los *hechos*, que se introducen a través de la relación `knowledge`, se refieren a hechos no incluidos en las ontologías, pero que son imprescindibles para relacionar las tareas con el dominio de la aplicación y las inferencias con los métodos. La Figura 4.16 muestra un ejemplo en el dominio de la fabricación que utiliza las ontologías denominadas *maquinas* y *operaciones* para describir las máquinas y las operaciones de fabricación, respectivamente. En este caso, la relación `knowledge` modela los hechos que se refieren a las operaciones concretas a realizar y a las máquinas específicas que las pueden llevar a cabo. Por ejemplo, la fórmula $realizaOperacion(canteadora, cantear)$ indica una relación binaria entre la máquina *canteadora* y la operación *cantear*. Además, el conocimiento apuntado por la relación `knowledge` también incluye reglas a partir de las cuales se pueden derivar nuevos hechos. Por ejemplo, la regla $realizaOperacion(m, cantear) \rightarrow realizaOperacion(m, lijar)$ indica que las máquinas que realizan un canteado también pueden utilizarse para el lijado.

DomainModel fabricación
pragmatics
Dominio de la fabricación de muebles.
ontologies
maquinas, operaciones
knowledge
Piezas del mueble a construir:
$pieza(listonDer)$
$longitud(listonDer, 200)$
$anchura(listonDer, 100)$
$espesor(listonDer, 20)$
$tieneRanura(listonDer, ranuraHorizontal)$
...
Relación entre máquinas y operaciones:
$realizaOperacion(canteadora, cantear)$
$realizaOperacion(seccionadora, seccionar)$
$realizaOperacion(centroMecanizado, macanizar)$
$realizaOperacion(robotMecanizado, macanizar)$
$realizaOperacion(fresadora, fresar)$
$realizaOperacion(robotMecanizado, fresar)$
...
Reglas para deducir nuevas relaciones máquina-operación:
$\forall m \ realizaOperacion(m, cantear) \rightarrow realizaOperacion(m, lijar)$
$\forall m \ realizaOperacion(m, mecanizar) \rightarrow realizaOperacion(m, taladrar)$
...
properties
$\forall o \, \exists m \ operacion(o) \rightarrow realizaOperacion(m, o)$
assumptions
A partir de las características, se saben los procesos de fabricación por los que ha de pasar una pieza:

$$\forall p, r \ pieza(p) \land procesar(p, mecanizar) \leftrightarrow tieneRanura(p, r) \lor (tieneTaladro(p, t) \land posicion(t, cara)) \lor (tieneRecorte(p, t) \land posicion(t, cara))$$

$$\forall p, r \ pieza(p) \land procesar(p, cantear) \leftrightarrow tieneCantoIzq(p, r) \lor tieneCantoDer(p, r) \lor tieneCantoInf(p, r) \lor tieneCantoSup(p, r) \lor$$

...

Figura 4.16: Ejemplo de modelo del dominio de fabricación en la industria del mueble

```
objectProperty(properties).
domain(properties, DomainModel).
range(properties, Formula).
```

La relación `properties` captura los teoremas que permiten establecer las *propiedades* que el dominio debe cumplir y que se suelen referir a elementos establecidos en el conocimiento del dominio. En la Figura 4.16 también se muestra conocimiento de este tipo: por ejemplo, la fórmula $\forall o \, \exists m \ operacion(o) \rightarrow realizaOperacion(m, o)$ indica que para cada operación existe una máquina que puede realizarla, es decir, todas

las operaciones pueden implementarse. Este tipo de relaciones es imprescindible para asegurar que una tarea/método se puede aplicar a un determinado dominio. Por ejemplo, esta relación evitaría que la ontología de *operaciones* tuviese alguna operación no soportada por, al menos, una máquina. Indirectamente, la relación asegura que si se aplica una tarea de planificación a un determinado dominio, el método que la resuelve podrá asignar una máquina a cada operación.

```
objectProperty(assumptions).
domain(assumptions, DomainModel).
range(assumptions, Formula).
```

Finalmente, el metaconocimiento se especifica a través de la relación `assumptions`. Estas fórmulas capturan las asunciones implícitas y explícitas del modelo construido. Se evaluarán siempre a *verdadero* y, por lo tanto, no es necesario probarlas. Estas asunciones son de gran utilidad, ya que permiten reducir el espacio de búsqueda de los procesos de razonamiento y suelen utilizarse para restringir algún aspecto del dominio que no puede derivarse a partir de otro conocimiento. Por ejemplo, las dos asunciones de la Figura 4.16 se utilizan para definir de forma completa las características que ha de tener una pieza para que se pueda realizar sobre ella una operación de mecanizado y de canteado.

4.4. Dimensión de recursos

Desde la perspectiva de los PSMs, los recursos no suelen modelarse. Aunque los SBCs pueden requerir la intervención humana o de agentes externos para la obtención de algún dato, este tipo de interacción no requiere un modelado explícito. Sin embargo, desde la perspectiva de los WFs, los recursos son una parte imprescindible, ya que toman parte activa en el proceso, llegando incluso a realizar alguna o todas las actividades del WF. Debido a estas dos visiones, durante el diseño del metamodelo surgió una pregunta: ¿cómo integrar los recursos dentro de un modelo orientado a la definición de PSMs? En esta tesis, se optó por la integración del *modelo de recursos como un componente de conocimiento* más del metamodelo.

Aunque durante el diseño de la arquitectura también se barajó la posibilidad de incluir a los recursos como una *parte del modelo del dominio*, esta aproximación se descartó por tres motivos:

- *Menor reutilización.* Al estar integrado dentro del modelo del dominio, la reutilización de la parte correspondiente a los recursos es complicada. Dependiendo del modo en el que se creó el modelo del dominio existen dos posibilidades: *(i)* el refinamiento, que sólo se aplicaría a los casos en los que el modelo del dominio hace *uso* (relación *uses*) de un modelo de recursos creado previa e independientemente, y *(ii)* la copia (manualmente) de la parte del modelo que interesa, y se aplicaría cuando los recursos están definidos dentro del modelo del dominio.

- *Puentes*. Aunque en WFs sencillos la integración de los recursos dentro del modelo del dominio podría funcionar a través de los puentes adecuados, si el WF se complica o implica a varias organizaciones esta solución deja de ser válida. Por ejemplo, en el caso que dos organizaciones pertenecientes a la misma institución estén habilitadas para la ejecución de un determinado método primitivo sería necesario integrar los modelos del dominio de ambas organizaciones. Esta solución es inviable, ya que el conocimiento en forma de reglas aplicable a una organización puede no serlo a la otra. En cambio, si los modelos de recursos y del dominio están separados este problema desaparece.

- *Semántica de ejecución*. El SBC que dé soporte a los WFs ha de ser capaz de interpretar el modelo de organización y también de planificar la mejor selección de recursos para cada método a ejecutar. Para ello, es necesario identificar los elementos del modelo de recursos con los que el planificador de la ejecución va a razonar. Si los recursos se incluyen como parte del modelo del dominio, su definición no estaría restringida a un determinado modelo (sería libre), con los que cada diseñador podría crear su propio modelo y usar sus propios criterios para clasificar los recursos. En esta situación sería necesario adaptar el planificador a cada modelo de recursos, lo cual es inviable desde un punto de vista más práctico.

4.4.1. Ontología de organización TOVE

Como se comentó en la sección 4.1.3, la dimensión de recursos utiliza el modelo de organización de la ontología TOVE para definir los agentes y especificar la estructura de la organización (empresa, institución, etc.) en la que se implanta el WF. La Figura 4.17 muestra la red semántica con las clases y relaciones más representativas de la ontología TOVE: una organización consiste en un conjunto de divisiones y (sub)divisiones, un conjunto de agentes que son miembros de alguna de las divisiones, un conjunto de roles que juegan los distintos agentes y un árbol de objetivos que los agentes y la organización intentan conseguir. Por ejemplo, el Departamento de Electrónica y Computación puede modelarse como una institución con unos objetivos de educación e investigación, divisiones como son el Grupo de Sistemas Inteligentes o el Laboratorio de Sistemas, un número de agentes como es el personal docente, de investigación o los propios estudiantes, y una serie de roles como el de profesor, investigador, secretario, etc. que dichos agentes pueden tomar.

Por otra parte, un agente puede tomar uno o más roles, y cada rol tiene a su vez asociado un conjunto de objetivos y una determinada autorización. Los agentes también tiene asignadas las actividades que pueden realizar en la organización y los recursos que esas actividades consumen (por ejemplo, el material, las herramientas, etc.). Finalmente, los agentes también se pueden agrupar en equipos para dar respuesta a una actividad especial.

Es importante señalar que nuestro modelo de recursos no incluirá todos los conceptos definidos en la ontología TOVE. Algunos conceptos no tienen una imagen en nuestro modelo debido a que son redundantes o innecesarios. Éste es el caso del

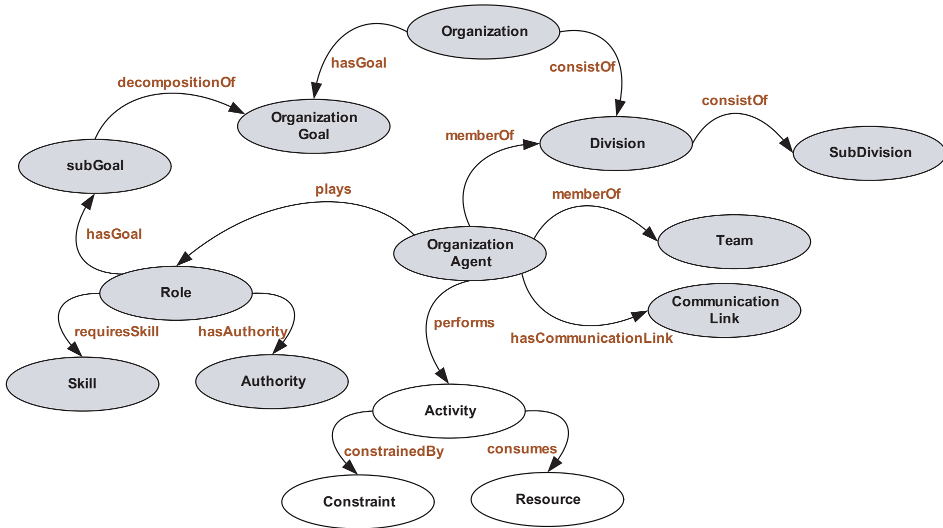


Figura 4.17: Red semántica con los principales elementos de la ontología de organización TOVE. Los conceptos coloreados en gris están incluidos en el modelo de recursos.

concepto `Activity` cuya semántica ya está recogida en el concepto `Task` de nuestro metamodelo.

4.4.2. Modelo de recursos

La clase `ResourceModel` se refiere al componente de conocimiento que permite capturar el modelo de recursos:

```
subClassOf(ResourceModel, KnowledgeComponent).

objectProperty(organizations).
domain(organizations, ResourceModel).
range(organizations, Organization).
```

Un modelo de recursos se define como un componente de conocimiento y se relaciona a través de la relación `organizations` con el conjunto de organizaciones asociadas al modelo. Esta relación tiene cardinalidad múltiple, ya que los WFs modelados por nuestro metamodelo también pueden ser darse entre distintas organizaciones.

```
objectProperty(agents).
domain(agents, ResourceModel).
range(agents, OrganizationalAgent).
```

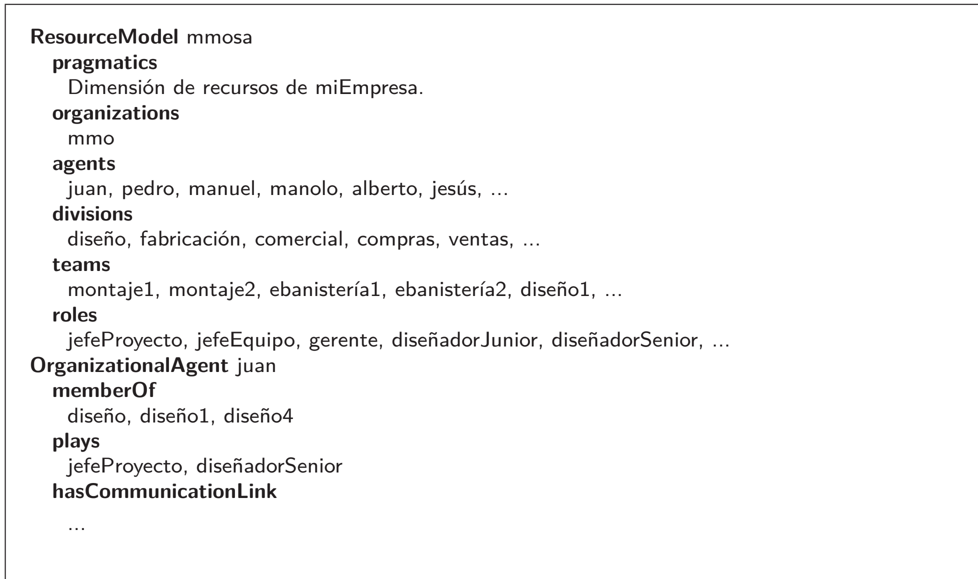

ResourceModel mmosa
pragmatics
Dimensión de recursos de miEmpresa.
organizations
mmo
agents
juan, pedro, manuel, manolo, alberto, jesús, ...
divisions
diseño, fabricación, comercial, compras, ventas, ...
teams
montaje1, montaje2, ebanistería1, ebanistería2, diseño1, ...
roles
jefeProyecto, jefeEquipo, gerente, diseñadorJunior, diseñadorSenior, ...
OrganizationalAgent juan
memberOf
diseño, diseño1, diseño4
plays
jefeProyecto, diseñadorSenior
hasCommunicationLink
...

Figura 4.18: Ejemplo de modelo de recursos

En cualquier caso, el principal objetivo de este modelo es clasificar y definir las características de los agentes que van a participar en la ejecución del WF y, para ello, la relación `agents` apuntará un conjunto de agentes descritos a través del concepto `OrganizationalAgent` de la ontología TOVE. En este sentido es necesario precisar que todos los conceptos de esta dimensión se referirán a clases de la ontología TOVE.

```
objectProperty(divisions).
domain(divisions, ResourceModel).
range(divisions, Division).

objectProperty(teams).
domain(teams, ResourceModel).
range(teams, Team).

objectProperty(roles).
domain(roles, ResourceModel).
range(roles, Role).
```

Las relaciones `divisions`, `teams` y `roles` se refieren a las distintas formas de *clasificar* a los agentes dentro del modelo de recursos. Como está representado en la Figura 4.17, estas clasificaciones se establecen a través de sus relaciones `memberOf` y `plays` que asocian las divisiones, equipos y roles al agente.

La Figura 4.18 representa un modelo de recursos de la empresa *mmosa* que está compuesto por varios agentes, divisiones, equipos, roles y actividades. Por ejemplo, en este modelo se incluyen los agentes *juan*, *pedro* o *manuel*; las divisiones de *diseño*,

fabricación o *compras*; algunos equipos de *montaje* o *ebanistería*; roles, como *jefeProyecto* o *diseñadorJunior*, que juegan los agentes en la ejecución; y algunas de las actividades que los agentes pueden ejecutar. Para facilitar la comprensión del modelo, la segunda parte del ejemplo muestra la definición del agente *juan*. En este caso, *juan* es miembro de la división *diseño*, de los equipos *diseño1* y *diseño4*, y juega el papel de *jefe de proyecto* y *diseñador senior* dentro del modelo de la organización.

4.5. Puentes

Los puentes modelan explícitamente la relación entre dos componentes de conocimiento *distintos* y son los responsables de que exista un fuerte desacoplamiento entre ellos. Consecuentemente, gran parte de la capacidad de reutilización del metamodelo de WFs recae en los puentes.

El metamodelo UPML define tres tipos de puentes a través de los cuales se enlaza una tarea con un método, una tarea con un modelo del dominio y un método con un modelo del dominio. Sin embargo, la introducción de los modelos de control y recursos en la arquitectura de UPML para adaptar el metamodelo a la definición de WFs han ampliado el número de puentes a siete. Estos puentes permiten relacionar:

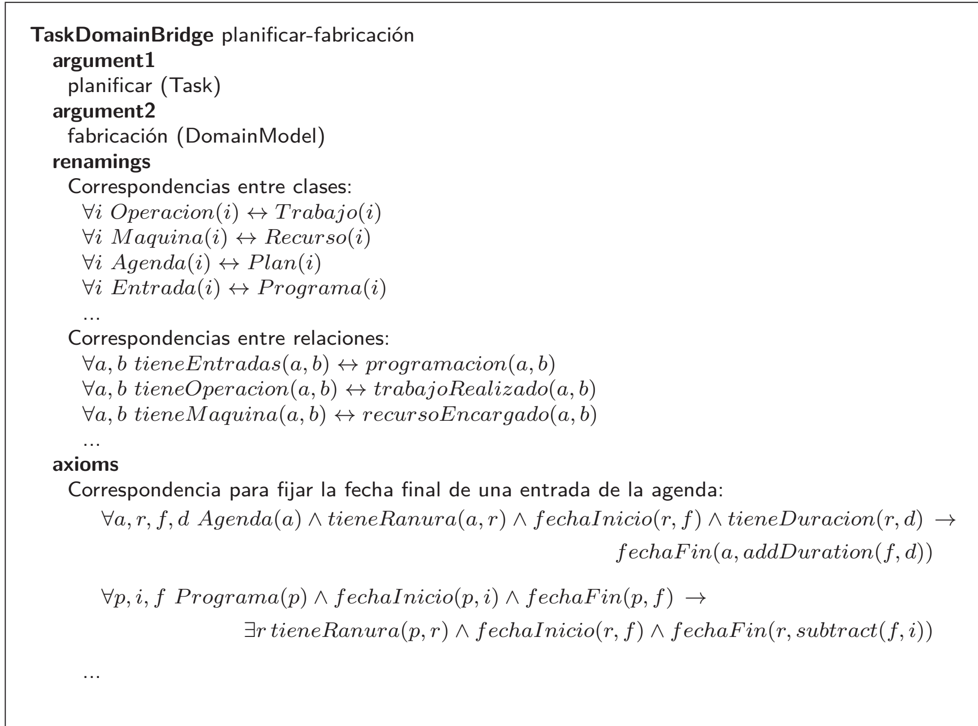
- Una tarea con un método.
- Una tarea con un modelo del dominio.
- Un método con un modelo del dominio.
- Un método con su modelo de control.
- Un método con su modelo de recursos.
- Un modelo del dominio con un modelo de recursos.
- Un modelo del dominio con un modelo de control.

Es importante resaltar que no existen puentes entre tareas y modelos de control y ni entre tareas y modelos de recursos. Este se debe a que los recursos y el control están directamente relacionados con la forma de ejecutar los métodos, lo cual en último término permite resolver la tarea.

```
subClassOf(Bridge, Adapter).

domain(axioms, Bridge).
domain(assumptions, Bridge).
```

La clase `Bridge` se refiere a este tipo de adaptador. La principal diferencia de los puentes respecto a los adaptadores genéricos son las fórmulas mediante las cuales pueden establecer axiomas y asunciones acerca de la correspondencias entre los componentes de concocimiento.



TaskDomainBridge planificar-fabricación
argument1
planificar (Task)
argument2
fabricación (DomainModel)
renamings
Correspondencias entre clases:
$\forall i\ Operacion(i) \leftrightarrow Trabajo(i)$
$\forall i\ Maquina(i) \leftrightarrow Recurso(i)$
$\forall i\ Agenda(i) \leftrightarrow Plan(i)$
$\forall i\ Entrada(i) \leftrightarrow Programa(i)$
...
Correspondencias entre relaciones:
$\forall a, b\ tieneEntradas(a, b) \leftrightarrow programacion(a, b)$
$\forall a, b\ tieneOperacion(a, b) \leftrightarrow trabajoRealizado(a, b)$
$\forall a, b\ tieneMaquina(a, b) \leftrightarrow recursoEncargado(a, b)$
...
axioms
Correspondencia para fijar la fecha final de una entrada de la agenda:

$$\forall a, r, f, d\ Agenda(a) \wedge tieneRanura(a, r) \wedge fechaInicio(r, f) \wedge tieneDuracion(r, d) \rightarrow fechaFin(a, addDuration(f, d))$$

$$\forall p, i, f\ Programa(p) \wedge fechaInicio(p, i) \wedge fechaFin(p, f) \rightarrow \exists r\ tieneRanura(p, r) \wedge fechaInicio(r, f) \wedge fechaFin(r, subtract(f, i))$$

...

Figura 4.19: Ejemplo de puente entre una tarea y un modelo del dominio

4.5.1. Puente Tarea-Dominio

Esta clase de puente se encarga de adaptar las tareas al dominio de la aplicación en el que se van a ejecutar. Más concretamente, el principal objetivo del puente es relacionar la terminología de la tarea con la del dominio para así poder dar una interpretación a las asunciones de las tareas. Por ejemplo, el puente representado en la Figura 4.19 muestra algunas correspondencias directas (*renamings*) entre la terminología de la tarea *planificar* y el modelo del dominio de *fabricación* descritos en apartados anteriores. En este puente el término trabajo se asocia con el de operación, el recurso con el de máquina, y así sucesivamente.

```
subClassOf(TaskDomainBridge, Bridge).

allValuesFrom(argument1, TaskDomainBridge, Task).
allValuesFrom(argument2, TaskDomainBridge, DomainModel).
```

En el caso de no poder establecer una correspondencia directa entre la terminología de ambos componentes, las equivalencias se establecen a través de axiomas. Por ejemplo, supóngase que se quieren definir las correspondencias entre las dos taxonomías de la

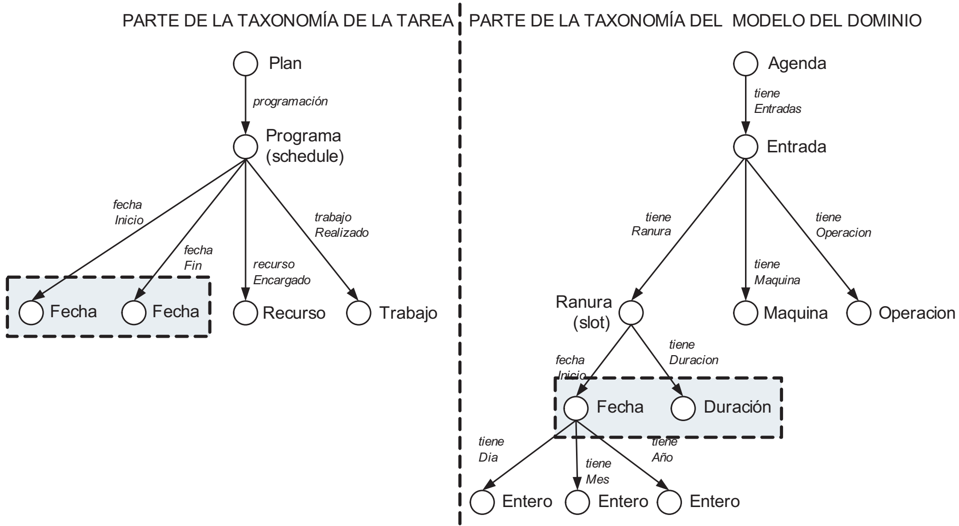


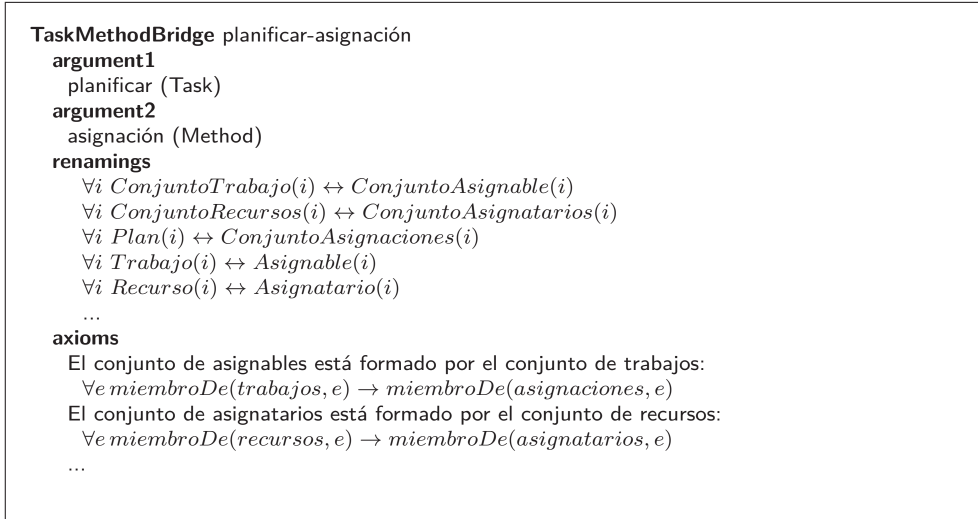
Figura 4.20: Para establecer las correspondencias entre los conceptos *Plan* y *Agenda* de las taxonomías de la tarea y del dominio respectivamente, es necesario además definir la axiomática que permita unificar los distintos modos de establecer la duración de un programa

Figura 4.20, que muestran parte de las ontologías usadas por la tarea (parte izquierda) y por el dominio (parte derecha). En este ejemplo existe una clara asimetría a la hora de fijar el tiempo y la duración: por un lado, la taxonomía de la tarea utiliza como ranura temporal las fechas de inicio y fin para establecer la duración; y, por otro lado, la taxonomía del dominio asocia una ranura temporal a la entrada de la agenda a partir de una fecha de inicio y una duración. Es necesario, por lo tanto, establecer la correspondencia entre la fecha de fin de la programación y la fecha de inicio y la duración de la agenda. El axioma representado en la Figura 4.19 captura esta correspondencia.

En ocasiones, en este tipo de puentes es necesario definir asunciones para asegurar que las correspondencias entre los dos componentes respetan la especificación de la tarea. Estas asunciones suelen ser un subconjunto de las asunciones formuladas en la tarea y que no están recogidas (es decir, no se cumplen) en las relaciones del modelo del dominio. En el caso del puente de la Figura 4.19 no es necesario definir ninguna asunción adicional entre la tarea *planificar* y el dominio de *fabricación*, ya que la única asunción definida en la tarea no afecta al modelo del dominio.

4.5.2. Puente Tarea-Método

La separación entre tarea y método posibilita la reutilización de un método para resolver distintas tareas y, a la inversa, la aplicación de diferentes métodos para una



TaskMethodBridge planificar-asignación
argument1
planificar (Task)
argument2
asignación (Method)
renamings
$\forall i\ ConjuntoTrabajo(i) \leftrightarrow ConjuntoAsignable(i)$
$\forall i\ ConjuntoRecursos(i) \leftrightarrow ConjuntoAsignatarios(i)$
$\forall i\ Plan(i) \leftrightarrow ConjuntoAsignaciones(i)$
$\forall i\ Trabajo(i) \leftrightarrow Asignable(i)$
$\forall i\ Recurso(i) \leftrightarrow Asignatario(i)$
...
axioms
El conjunto de asignables está formado por el conjunto de trabajos:
$\forall e\ miembroDe(trabajos, e) \rightarrow miembroDe(asignaciones, e)$
El conjunto de asignatarios está formado por el conjunto de recursos:
$\forall e\ miembroDe(recursos, e) \rightarrow miembroDe(asignatarios, e)$
...

Figura 4.21: Ejemplo de puente entre una tarea y un método

misma tarea. La conexión entre tareas y métodos se establece a través del concepto `TaskMethodBridge` y permite establecer las correspondencias entre ambos componentes:

```
subClassOf(TaskMethodBridge, Bridge).

allValuesFrom(argument1, TaskMethodBridge, Task).
allValuesFrom(argument2, TaskMethodBridge, Method).
```

Por ejemplo, en la Figura 4.21 se puede apreciar cómo alguna de estas correspondencias son directas, como las que existen entre el conjunto de *trabajos* (especificado en la tarea) y el conjunto de elementos *asignables* (especificado en el método), mientras que otras necesitan de axiomas. Ambos axiomas aseguran que los conjuntos asignables y asignatarios (asociados al método) se componen de los mismos elementos que los conjuntos de entrada de la tarea.

Es importante resaltar que este puente, unido al puente entre la tarea y el dominio, posibilitan que exista una cadena de correspondencias que permite asociar los datos del dominio con las definiciones del método.

4.5.3. Puente Método-Dominio

La principal función de este puente es establecer las correspondencias entre las terminologías empleadas en los métodos y en el modelo del dominio. Sin embargo, en la mayoría de los casos estas correspondencias ya están establecidas de forma transitiva

a través de los puentes que relacionan a los métodos con las tareas y a las tareas con el modelo del dominio. Por este motivo, este puente suele establecer únicamente aquellas correspondencias que no están cubiertas por los otros dos puentes: suele tratarse de requerimientos acerca del conocimiento usado para el proceso de razonamiento que, por lo tanto, no están recogidos en las tareas. El puente `MethodDomainBridge` se encarga de hacer compatibles estos requerimientos con el conocimiento heurístico del dominio.

```
subClassOf(MethodDomainBridge, Bridge).

allValuesFrom(argument1, MethodDomainBridge, Method).
allValuesFrom(argument2, MethodDomainBridge, DomainModel).
```

4.5.4. Puente Método-Control

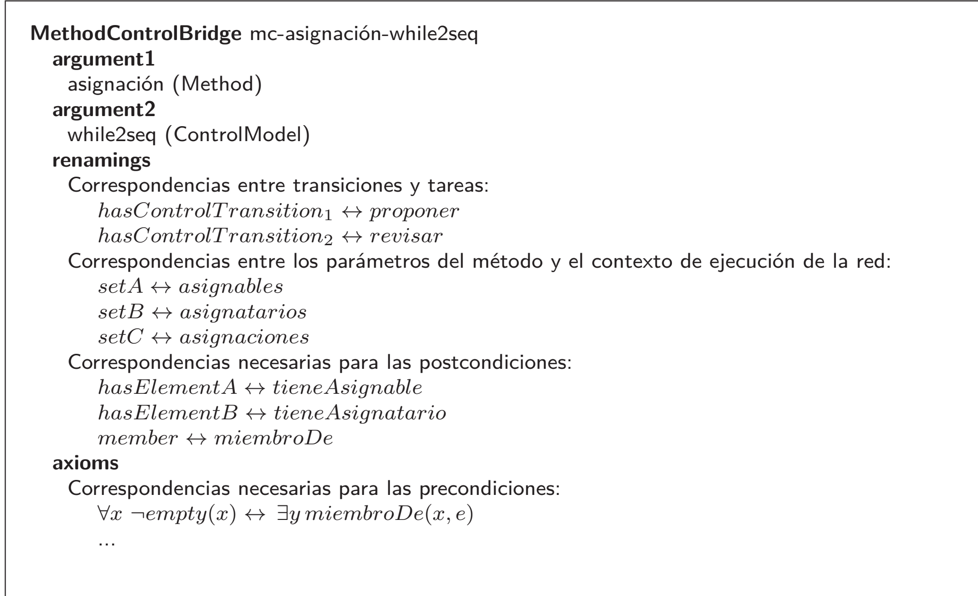
Al desligar los métodos de su descripción operacional y crear un nuevo elemento en la arquitectura para este propósito, se hace necesario definir un nuevo puente para relacionar ambos componentes. Estas correspondencias sirven para relacionar la firma de la red de Petri a la terminología definida por el método: el objetivo de este puente es, por lo tanto, que algunos de los operadores que anotan las condiciones de guardia de las transiciones y los arcos de la red obtengan su semántica a partir de la terminología del método. Es importante mencionar que no todos los operadores empleados para anotar una red van a tener una correspondencia definida con el método por medio de este puente. Algunos de estos operadores obtendrán su interpretación a partir del modelo del dominio empleado por el WF. Sin embargo, todas las características del método, es decir, sus entradas, salidas, precondiciones y postcondiciones, *sí* estarán asociados a algún elemento del modelo de control.

```
subClassOf(MethodControlBridge, Bridge).

allValuesFrom(argument1, MethodControlBridge, Method).
allValuesFrom(argument2, MethodControlBridge, ControlModel).
```

La Figura 4.22 recoge las correspondencias definidas entre el método de asignación de recursos descrito en la Figura 4.10 y la red de control representada en la Figura 4.23. Un aspecto interesante de este ejemplo son las correspondencias que se establecen entre algunas de las transiciones y las (sub)tareas del método. Las dos primeras correspondencias se refieren a esta característica del modelo y recogen la asociación establecida entre las transiciones $hasControlTransition_1$ y $hasControlTransition_2$ y las tareas *proponer* y *revisar*, respectivamente. Como se puede ver en la representación gráfica del control de la Figura 4.14, estas correspondencias indican que las dos tareas estarán dentro de la *secuencia* incluida en el bucle *repetir-mientras*.

Como se comentará en el Capítulo 5 y se puede ver en la Figura 4.23, las plazas de las redes de Petri que modelan la descripción operacional están anotadas con un único tipo que representa el *contexto* de ejecución del método. Este tipo, denominado `Context`, ha de contener una entrada por cada uno de los parámetros del método,



MethodControlBridge mc-asignación-while2seq
argument1
asignación (Method)
argument2
while2seq (ControlModel)
renamings
Correspondencias entre transiciones y tareas:
$hasControlTransition_1 \leftrightarrow proponer$
$hasControlTransition_2 \leftrightarrow revisar$
Correspondencias entre los parámetros del método y el contexto de ejecución de la red:
$setA \leftrightarrow asignables$
$setB \leftrightarrow asignatarios$
$setC \leftrightarrow asignaciones$
Correspondencias necesarias para las postcondiciones:
$hasElementA \leftrightarrow tieneAsignable$
$hasElementB \leftrightarrow tieneAsignatario$
$member \leftrightarrow miembroDe$
axioms
Correspondencias necesarias para las precondiciones:
$\forall x\ \neg empty(x) \leftrightarrow\ \exists y\, miembroDe(x, e)$
...

Figura 4.22: Ejemplo de puente entre un método y un modelo de control

independientemente de si son de entrada o de salida. Además, las correspondencias deben de relacionar las anotaciones de la red de Petri con las entradas, salidas, y pre/postcondiciones del método. En la Figura 4.22, la mayoría de estas correspondencias son directas, como, por ejemplo $setA \leftrightarrow asignables$, que permite establecer la relación entre el rol de entrada *asignables* del método y el operador *setA* de la firma sintáctica del modelo de control.

Finalmente, el último tipo de correspondencia recogida en el puente método-control traslada las precondiciones y postcondiciones del método a la semántica operacional de la red de control. Como se ha comentado en el Capítulo 3, las HL-PNs se anotan mediante una firma sintáctica para así permitir varias interpretaciones de una misma red. Este principio es adoptado por los modelos de control, de forma que su definición no incorpora el álgebra necesaria para dotar a dichas redes de una semántica operacional. Por ejemplo, los operadores *setA* o *empty* de la condición $Context(x)$ *and* $hasParameter(x, a)$ *and* $setA(a)$ *and not* $empty(a)$ *and* $hasParameter(x, b)$ *and* $setB(b)$ *and not* $empty(b)$ que anota la transición *processIf* en la Figura 4.23 no tienen una semántica operacional. Esta semántica se establece a través de las correspondencias con el método, de modo que la condición es equivalente a $Context(x)$ *and* $hasParameter(x, a)$ *and* $asignables(a)$ *and* $(\exists y\, miembroDe(a, y))$ *and* $hasParameter(x, b)$ *and* $asignatarios(b)$ *and* $(\exists y\, miembroDe(b, y))$. Puede comprobarse también que esta condición es equivalente a la precondición establecida para el método *asignación*.

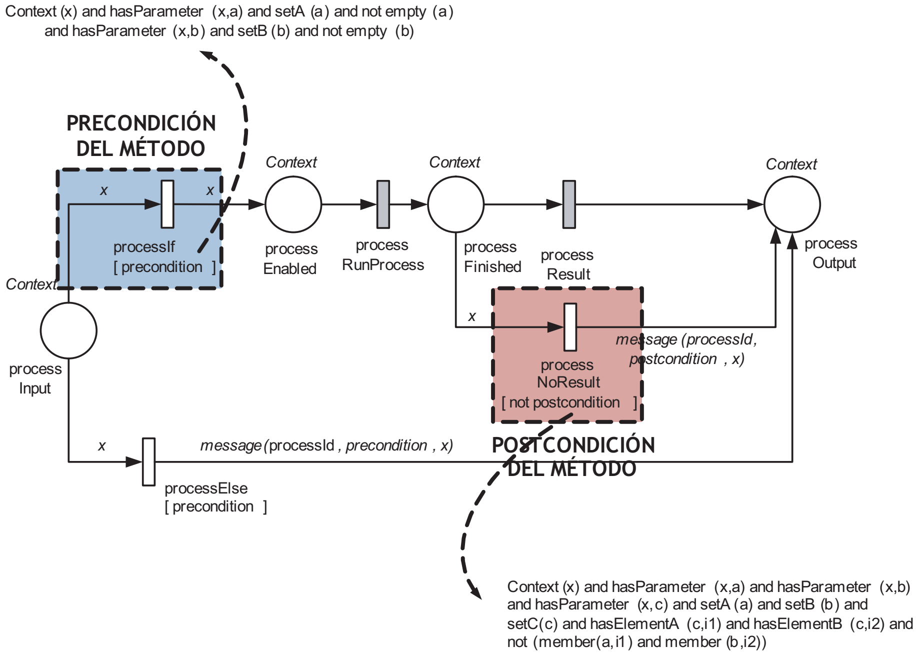


Figura 4.23: Parte de la descripción operacional del método descrito en la Figura 4.10

4.5.5. Puente Método-Recursos

Este puente *(i)* unifica la terminología entre los métodos y los modelos de recursos, además de *(ii)* indicar los criterios de asignación de trabajo que se usarán a la hora de seleccionar los agentes que ejecutarán el método.

```
subClassOf(MethodResourceBridge, Bridge).

allValuesFrom(argument1, MethodResourceBridge, ReasoningResource).
allValuesFrom(argument2, MethodResourceBridge, ResourceModel).
```

La política de asignación de trabajo se establece a través de la relación `assignmentCriteria`, que indica los criterios que regirán la selección de los recursos más apropiados para llevar a cabo la ejecución del método. Como se ha detallado en el Capítulo 2 existen muchos posibles criterios de asignación de trabajo, y en este metamodelo se han implementado dos de ellos:

- *Criterios de asignación automáticos.* La clase `AutomaticCriteria` permite establecer el tipo de asignación automática a realizar. En función del valor que tome la relación `assignmentType`, la asignación estará basada en el historial de trabajo del agente, en los casos que haya resuelto previamente o en su carga de trabajo.

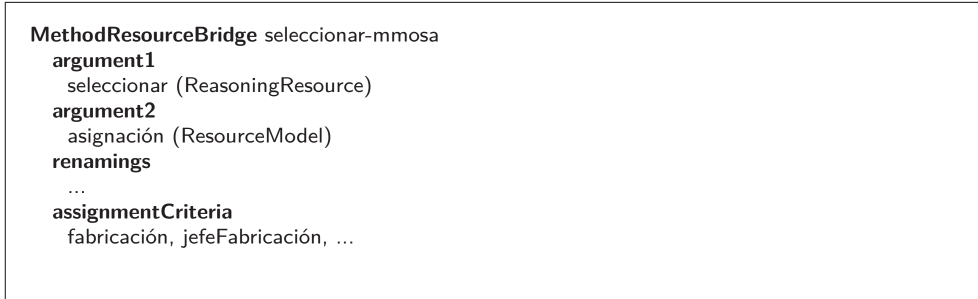


Figura 4.24: Ejemplo de puente entre un método y un modelo de recursos

- *Criterios de asignación basados en la estructura organizativa.* La clase `DefinedCriteria` se refiere a los criterios que se establecen a partir de elementos del modelo de recursos, es decir, divisiones, equipos, roles, agentes, habilidades y autorizaciones. En el ejemplo de la Figura 4.24 se puede observar como la relación `assignmentCriteria` toma los valores *fabricación* y *jefeFabricación*. Esto significa que el método *seleccionar* se asignará preferentemente a aquellos agentes que formen parte de la división de fabricación y que jueguen el papel de jefes de fabricación.

```
objectProperty(assignmentCriteria).
domain(assignmentCriteria, MethodResourceBridge).
range(assignmentCriteria, Criterion).

subClassOf(AutomaticCriteria, Criterion).
objectProperty(assignmentType).
domain(assignmentType, MethodResourceBridge).
oneOf(assignmentType, [historicalBased, caseBased, workloadBased]).

subClassOf(DefinedCriteria, Criterion).
union(DefinedCriteria, [Division, Team, Role, OrganizationalAgent, Skill, Authority]).
```

En la Figura 4.24 se muestra un ejemplo de este tipo de puente que relaciona al método primitivo *seleccionar* con el modelo de recursos *miEmpresa* (representados respectivamente en las figuras 4.11 y 4.18). En este ejemplo se indica que los agentes habilitados para la ejecución del método *seleccionar* serán aquellos que pertenezcan a la división de *fabricación* y que tengan el rol de *jefeFabricación*.

4.5.6. Puente Dominio-Control

Este puente permite completar la semántica operacional del modelo de control a partir de los elementos pertenecientes al modelo del dominio. El concepto `DomainControlBridge` se refiere a este tipo de puente:

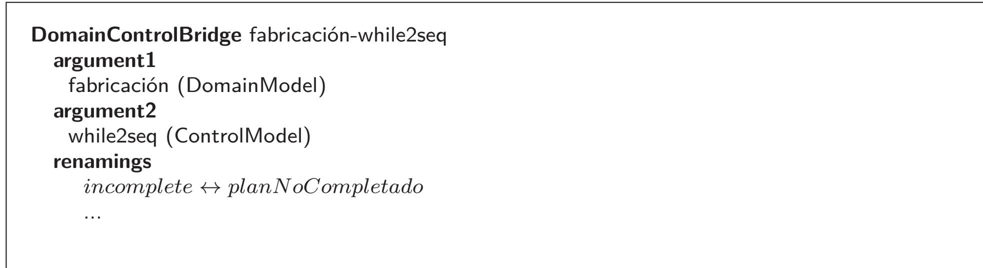


Figura 4.25: Ejemplo de puente entre un modelo del dominio y un modelo de control

```
subClassOf(DomainControlBridge, Bridge).

allValuesFrom(argument1, DomainControlBridge, Domain).
allValuesFrom(argument2, DomainControlBridge, ControlModel).
```

Como se comentó en el Capítulo 3 y se analizará más en profundidad en el Capítulo 5, las HLPNs se anotan mediante una firma sintáctica compuesta por tipos de datos y operadores, pero no tienen asociada ninguna semántica operacional. Para dotar de semántica a estos elementos es necesario relacionar cada uno de ellos con un elemento del dominio de la aplicación.

La mayoría de las correspondencias entre el modelo de control y el modelo del dominio ya están establecidas transitivamente a través de otros puentes del metamodelo. Por ejemplo los elementos del contexto de ejecución del modelo de control se asocian a los roles de entrada y salida del método (puente método-control), los cuales a su vez se relacionan con su interpretación en el dominio de la aplicación (puente método-dominio o método-tarea-dominio). Por ello en este puente solamente se definirán aquellas correspondencias que no están incluidas en los otros puentes, y que se refieren a anotaciones de las HLPNs cuyos operadores no tienen asociada ninguna interpretación.

Para el modelo de control representado en la Figura 4.14 la única correspondencia que falta por establecer se refiere al operador *incomplete* que aparece en la condición de las transiciones *whileIf* y *whileElse* de la HLPN *WhilePage*. Este operador se define dentro de la firma sintáctica como $incomplete : setC- > Boolean$ y, como se puede apreciar en la Figura 4.25, será sinónimo del término *planNoAceptado* del modelo del dominio.

4.5.7. Puente Dominio-Recursos

Este tipo de puentes permite relacionar la terminología empleada para describir al modelo de la organización con la terminología del dominio de la aplicación. Las correspondencias definidas por este puente suelen referirse a las funcionalidades y ha-

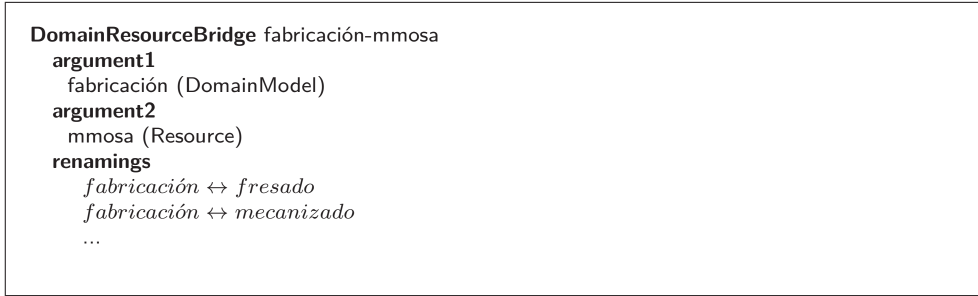


Figura 4.26: Ejemplo de puente entre un modelo del dominio y un modelo de recursos

bilidades de los recursos, las cuales, en ocasiones, también están representadas en el dominio.

```
subClassOf(DomainResourceBridge, Bridge).

allValuesFrom(argument1, DomainResourceBridge, Domain).
allValuesFrom(argument2, DomainResourceBridge, ResourceModel).
```

La Figura 4.26 muestra un ejemplo de puente entre estos dos componentes, en el que el término *fabricación* utilizado para describir una habilidad de un rol se corresponde con los términos *fresado* y *mecanizado* del dominio de la aplicación. Este tipo de correspondencias pueden parecer a simple vista innecesarias: ¿qué necesidad existe de indicar estos sinónimos? La razón es que debido a estas relaciones y a la capacidad de razonamiento de un marco basado en el conocimiento, se pueden deducir nuevas funcionalidades o habilidades acerca de los recursos.

4.6. Refinadores

Al contrario que los puentes, los refinadores conectan componentes de conocimiento del *mismo tipo*. Su función es crear un componente nuevo refinando o extendiendo alguna de las propiedades/características de un componente de conocimiento original. El uso de este tipo de adaptadores es otra de las facilidades de reutilización del metamodelo que es de mucha utilidad en el dominio de los WFs. Habitualmente, las empresas disponen de procesos de gestión/administrativos que suelen sufrir modificaciones ante la irrupción de alguna funcionalidad/trámite. Por ello, es fundamental disponer de un mecanismo que permita reutilizar el WF original, añadiendo los cambios sin necesidad de crearlo todo de nuevo.

```
subClassOf(Refiner, Adapter).
```

El metamodelo dispone de una clase de refinador por cada uno de los componentes de conocimiento; es decir, hay refinadores de tareas, métodos, modelos de control, modelos de dominio, modelos de recursos y ontologías.

4.6.1. Refinador de tarea

Este refinador se utiliza para crear una nueva tarea que restrinja o modifique alguna de las características de la tarea referida en la relación `argument1`. Por ejemplo, con este adaptador es posible añadir nuevos roles de entrada y de salida a las tareas. Sin embargo, no tendría sentido eliminar roles, ya que las asunciones heredadas de la tarea original podrían quedar inconsistentes.

```
subClassOf(TaskRefiner, Refiner).

allValuesFrom(argument1, TaskRefiner, Task).
allValuesFrom(argument2, TaskRefiner, Task).

domain(inputRoles, TaskRefiner).
domain(outputRoles, TaskRefiner).
domain(competence, TaskRefiner).
domain(assumptions, TaskRefiner).
```

Este tipo de refinadores también permite modificar la competencia (pre- y postcondiciones) y las asunciones de la tarea. Cuando se refina una precondición se están restringiendo los valores que pueden tomar los roles de entrada de la tarea . De la misma forma, la modificación de las postcondiciones afecta a los valores de las salidas. En cuanto a las asunciones, añadir nuevas asunciones reduce el número de estados que la tarea podría tener que procesar para alcanzar sus objetivos: al asumir ciertos hechos como ciertos, no es necesario probarlos durante el proceso de razonamiento.

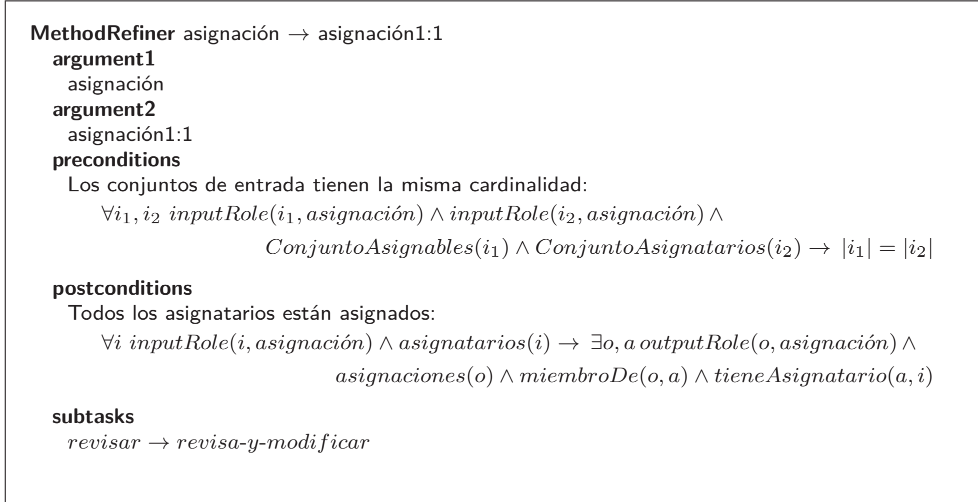
4.6.2. Refinador de método

Este adaptador permite refinar un método restringiendo sus entradas, salidas, y pre/postcondiciones. Tal y como sucede en el refinador de tareas no se pueden eliminar roles debido a que las condiciones heredadas podrían no tener sentido o quedar inconsistentes. Estas características son comunes a los refinadores de los métodos compuestos y de métodos primitivos.

```
subClassOf(MethodRefiner, Refiner).

allValuesFrom(argument1, MethodRefiner, Method).
allValuesFrom(argument2, MethodRefiner, Method).

domain(inputRoles, MethodRefiner).
domain(outputRoles, MethodRefiner).
domain(competence, MethodRefiner).
```



MethodRefiner asignación → asignación1:1
argument1
asignación
argument2
asignación1:1
preconditions
Los conjuntos de entrada tienen la misma cardinalidad:
$\forall i_1, i_2 \; inputRole(i_1, asignación) \wedge inputRole(i_2, asignación) \wedge$
$ConjuntoAsignables(i_1) \wedge ConjuntoAsignatarios(i_2) \rightarrow \; |i_1| = |i_2|$
postconditions
Todos los asignatarios están asignados:
$\forall i \; inputRole(i, asignación) \wedge asignatarios(i) \rightarrow \; \exists o, a \; outputRole(o, asignación) \wedge$
$asignaciones(o) \wedge miembroDe(o, a) \wedge tieneAsignatario(a, i)$
subtasks
revisar → *revisa-y-modificar*

Figura 4.27: Ejemplo de refinador que especializa el método compuesto representado en la Figura 4.10

Los métodos compuestos se refinan a través de la clase `ProblemDecomposerRefiner` con el que se modifican las (sub)tareas que componen un método. Del mismo modo que no se pueden eliminar entradas o salidas de una tarea/método, tampoco se pueden eliminar (sub)tareas. Por lo tanto, este tipo de refinador sólo se permite *intercambiar* o *añadir* nuevas tareas a la descomposición del método. Por ejemplo, la Figura 4.27 muestra un refinador que añade una precondición, una postcondición, e intercambia una de las tareas que componen el método de asignación. La nueva precondición incorpora la restricción de que los conjuntos de elementos asignables y asignatarios tienen la misma cardinalidad. Asimismo, la postcondición también restringe el tamaño del conjunto de asignaciones, ya que asume que todos los asignatarios están asignados.

```
subClassOf(ProblemDecomposerRefiner, MethodRefiner).

allValuesFrom(argument1, ProblemDecomposerRefiner, ProblemDecomposer).
allValuesFrom(argument2, ProblemDecomposerRefiner, ProblemDecomposer).

domain(subTasks, ProblemDecomposerRefiner).
```

Los métodos que se resuelven a través a través de primitivas de razonamiento se refinan mediante la clase `ReasoningResourceRefiner`. Esta clase permite añadir nuevos roles de conocimiento (roles estáticos) y más asunciones que se podrían aplicar sobre estos roles o sobre los pertenecientes al método original (roles de entrada y salida).

```
subClassOf(ReasoningResourceRefiner, MethodRefiner).

allValuesFrom(argument1, ReasoningResourceRefiner, ReasoningResource).
```

```
allValuesFrom(argument2, ReasoningResourceRefiner, ReasoningResource).

domain(knowledgeRoles, ReasoningResourceRefiner).
domain(assumptions, ReasoningResourceRefiner).
```

4.6.3. Refinador del modelo de control

La modificación de un modelo de control se realiza a través de este refinador, con el que se puede: modificar la firma de la red para añadir o cambiar operadores y tipos de datos, modificar las páginas de la red jerárquica incorporando nuevas HLPNs que describen alguna nueva construcción de control, y añadir algún nuevo mecanismo de composición para unir las nuevas HLPNs con las páginas ya existentes en la red. Sin embargo, al igual que en los demás refinadores, no se permite el borrado. En este sentido, no tendría sentido borrar elementos de la firma o de la estructura de la red jerárquica, ya que con ello el objeto refinado tendría un mayor universo de discurso que el objeto original, lo cual es inconsistente con el propósito de cualquier refinador.

```
subClassOf(ControlRefiner, Refiner).

allValuesFrom(argument1, ControlRefiner, Control).
allValuesFrom(argument2, ControlRefiner, Control).

domain(hasSignature, ControlRefiner).
domain(hasPages, ControlRefiner).
domain(hasSubstitutions, ControlRefiner).
domain(hasFusions, ControlRefiner).
```

En la práctica es muy habitual encontrarse con una red, cuya estructura sea muy parecida a la deseada. En estos casos, si el comportamiento deseado no puede conseguirse sustituyendo o fusionando alguno de los elementos de la red, es mejor crear una nueva red común y adaptar su comportamiento. Por ejemplo, la Figura 4.28 muestra gráficamente esta forma de proceder. La parte de la izquierda representa a la red original no adaptable y la parte de la derecha la adaptación del nuevo escenario donde *(i)* se ha creado una red común y *(ii)* se ha refinado esta red para dar soporte a la red original y a la red con un nuevo comportamiento, incorporando en sendas redes una nueva sustitución a la red jerárquica.

4.6.4. Refinador del modelo del dominio

Este tipo de refinador permite especificar un nuevo modelo del dominio a partir de uno ya existente. Los refinadores del dominio son muy utilizados, ya que permiten añadir nuevo conocimiento al ya existente en la relación `knowledge`. Por ejemplo, si se partiese del modelo descrito en la Figura 4.16, sería sencillo añadir nuevas máquinas, operaciones y relaciones al nuevo modelo. También permiten incorporar nuevas relaciones y asunciones adicionales para restringir el espacio de búsqueda a la hora de realizar inferencias.

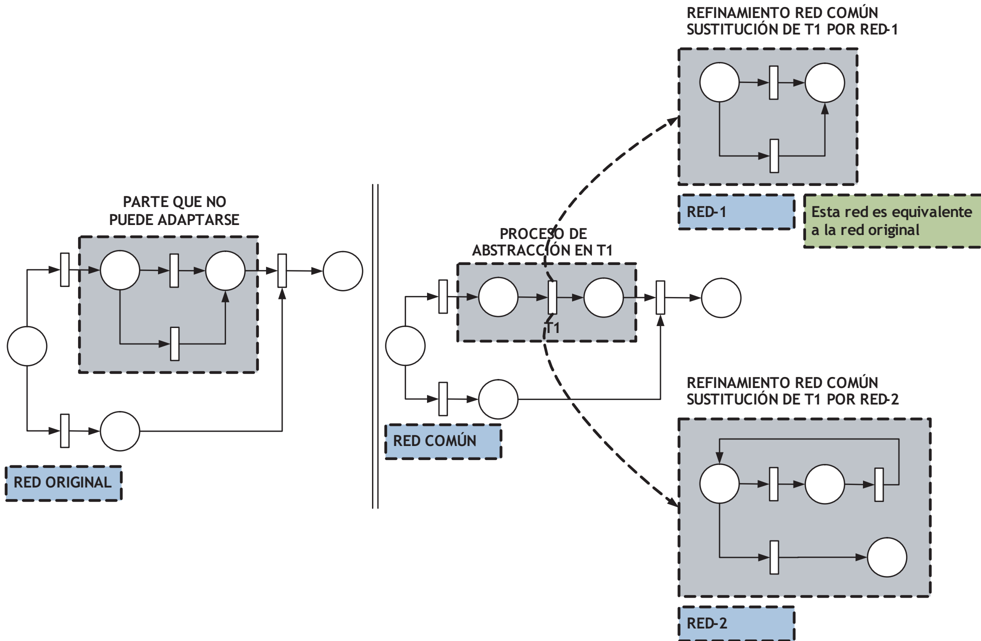


Figura 4.28: Ejemplo de puesta en común de un modelo de control para así aprovechar mejor la capacidad de reutilización de los refinadores

```
subClassOf(DomainRefiner, Refiner).

allValuesFrom(argument1, DomainRefiner, Domain).
allValuesFrom(argument2, DomainRefiner, Domain).

domain(knowledge, DomainRefiner).
domain(properties, DomainRefiner).
domain(assumptions, DomainRefiner).
```

4.6.5. Refinador del modelo de recurso

El refinador del modelo de recursos facilita la creación de un nuevo modelo organizativo a partir de uno ya existente. Este tipo de refinador es muy similar al refinador del dominio (tienen las mismas relaciones) y únicamente se diferencia de este último en el tipo de conocimiento manejado. Por lo tanto, a través de este refinador se podrán incluir nuevos recursos, divisiones, equipos, roles, responsabilidades, reglas de asignación, etc.

```
subClassOf(ResourceRefiner, Refiner).

allValuesFrom(argument1, ResourceRefiner, Resource).
allValuesFrom(argument2, ResourceRefiner, Resource).
```

```
domain(knowledge, ResourceRefiner).
domain(properties, ResourceRefiner).
domain(assumptions, ResourceRefiner).
```

4.6.6. Refinador de ontologías

Un último tipo de refinador es el que afecta a las ontologías, que como cualquier otro componente de conocimiento, también se pueden refinar de cara a facilitar su reutilización. El refinado de las ontologías se centra en:

- *Firma* (taxonomía). Adición de nuevos elementos a la firma, es decir, nuevas clases, relaciones, individuos, etc. En este caso el principal mecanismo de refinamiento es la herencia dado que con ella podemos añadir de forma sencilla cualquier elemento de la taxonomía.
- *Axiomas*. Adición de nuevos axiomas que restrinjan la semántica de las clases, relaciones e individuos de la ontología.

```
subClassOf(OntologyRefiner, Refiner).

allValuesFrom(argument1, OntologyRefiner, Ontology).
allValuesFrom(argument2, OntologyRefiner, Ontology).

domain(signature, ResourceRefiner).
domain(axioms, ResourceRefiner).
```

4.7. Conclusiones

En este capítulo hemos descrito en profundidad la propuesta arquitectónica del metamodelo presentado en la tesis doctoral. Esta arquitectura está pensada para modelar WFs desde una perspectiva orientada al conocimiento, de forma que exista un fuerte desacoplamiento entre sus componentes y se facilite su reutilización. Con ello no se pretende romper con la forma tradicional de modelar un WF, sino complementar esta visión con las ventajas que aporta una adecuada gestión del conocimiento. Un metamodelo que rompiese con el modelado tradicional de WFs no tendría sentido, ya que el éxito de esta tecnología radica en la forma intuitiva en la que se representan tanto el control del proceso como la estructura organizativa detrás de su ejecución. En este sentido, el metamodelo pretende formalizar y enriquecer los elementos que participan en la definición de un WF, pero no cambiar la filosofía de diseño de estos procesos: la idea es que cada una de las dimensiones que conforman un WF pueda definirse como un componente de conocimiento, y que dicho componente sea *fácilmente* reutilizable (además de permitir razonar acerca de sus características).

Nuestra aproximación se fundamentó en UPML, un metamodelo pensado para el diseño de SBCs mediante PSMs. El diseño de WFs mediante UPML tiene una importante ventaja respecto al marco tradicional en el que habitualmente se modelan:

la representación y gestión del conocimiento. Como hemos comentado en el Capítulo 1 los sistemas tradicionales integran el control con los datos, lo cual prácticamente impide su reutilización. El metamodelo presentado en este capítulo hace *explícita esta separación* de forma que el conocimiento dinámico de los WFs se describe independientemente de los dominios en los que puede aplicarse. Sin embargo, la adaptación de UPML es necesaria, ya que adolece de dos inconvenientes importantes (Tabla 4.1) al modelado de WFs:

- *Representación de los procesos*. UPML permite capturar el conocimiento dinámico utilizado para describir la estructura de los procesos y, consecuentemente, hacer explícito y reutilizable dicho conocimiento. Específicamente, su metamodelo representa los procesos haciendo uso de tres componentes: tareas, métodos y modelos del dominio. Las tareas describen desde una perspectiva funcional los objetivos, entradas, salidas, condiciones de aplicabilidad y asunciones asociadas al problema a resolver. Los métodos se refieren a la forma de resolver las tareas y, por lo tanto, aportan el conocimiento dinámico sobre cómo organizar la solución al problema. Finalmente, los modelos del dominio permiten especificar los hechos, reglas, axiomas y asunciones que caracterizan el conocimiento del dominio de aplicación. Sin embargo, UPML no permite capturar adecuadamente muchos de los detalles de control habituales en los WFs: los lenguajes utilizados para describir la operacionalidad de los métodos, como CML2 u OCML, *(i)* no son lo suficientemente expresivos para representar muchos de los patrones de WFs vistos en el Capítulo 2, y *(ii)* no son lo suficientemente legibles para que los diseñadores puedan manejar WFs con la fluidez en que acostumbran a utilizar los lenguajes visuales en los que se modelan los WFs.

- *Representación y coordinación de los participantes*. UPML no modela de forma explícita los recursos que participan en la ejecución de los métodos y, por lo tanto, no posee capacidades de coordinación: aunque un recurso podría incluirse en el modelo del dominio, UPML no le asocia una semántica en la ejecución del método ni permite asociarle criterios de asignación de trabajo. En este sentido, y al contrario de otras propuestas, como por ejemplo el modelo de agentes de CommonKADS, UPML simplemente no modela los ejecutores de los métodos.

Por estos motivos, en el metamodelo propuesto la arquitectura de UPML ha sido modificada con la inclusión de dos nuevos componentes:

- *Modelo de control*. Este modelo describe semánticamente la estructura de los WFs, y facilita su reutilización independientemente de la definición de los métodos. El modelo de control se centra en la explicitación de las estructuras de comportamiento (secuencias, separaciones, uniones, sincronizaciones, condicionales, etc.) que tanto éxito han tenido en el modelado de WFs. Estas estructuras están descritas a su vez a través de un formalismo de procesos aceptado dentro del ámbito investigador e industrial de los WFs: las redes de Petri de alto nivel

Tabla 4.1: Ventajas y desventajas de UPML y OPENET4WF (nuestra aproximación) para el modelado de WFs. Los signos '+' indican un soporte directo, los '+/-' un soporte parcial o incompleto y los '-' que no soporta la característica indicada. Es necesario precisar que un '-' no implica que sea imposible representar la característica indicada, sino que la solución no es directa o adecuada.

	UPML	OPENET4WF
Modelado de las características de los WFs		
Funcionalidad	+	+
Comportamiento	+/-	+
Tratamiento de la información	+	+
Operacionalidad	+/-	+
Organización	-	+
Reutilización del conocimiento		
Proceso	+	+
Dominio	+	+
Recursos	-	+
Razonamiento		
Modelo procesable	+	+

que están especificadas dentro del metamodelo a través de las ontologías de redes de Petri descritas en el Capítulo 3. De esta forma, el metamodelo propuesto preserva el legado expresivo y de representación gráfica de los WFs.

- *Modelo de recursos*. Este modelo captura la dimensión de recursos de los WFs a partir de la cual se establecerá la política de asignación de trabajos. Cada modelo ha de proporcionar tres aspectos básicos del modelo organizativo. Primero, ha de organizar los recursos que participan en el WF en divisiones, equipos, departamentos u otras unidades organizativas. Segundo, ha de establecer los papeles que cada uno de los recursos puede jugar dentro del la ejecución del WF y, si es necesario, las habilidades que posee y las políticas de seguridad a aplicar a cada uno de esos roles. Finalmente, ha de establecer las actividades que cada uno de los recursos está capacitado para ejecutar. A partir de estos elementos, las reglas de asignación de trabajo establecidas en este mismo modelo permitirán automatizar la selección del recurso más adecuado.

La unión entre los distintos componentes de conocimiento que forman la arquitectura también se modificó con el fin de adaptar UPML al nuevo escenario. En particular, la arquitectura añade cuatro puentes (*bridges*) para relacionar los modelos de control y de recursos con los métodos y el modelo del dominio. Teniendo en cuenta esta visión del modelo, se puede definir un WF como una configuración de un conjunto de componentes de conocimiento unidos a través de un conjunto de puentes. Desde la perspectiva del modelado de WFs las ventajas de nuestra aproximación respecto a UPML son por ello importantes y están resumidas en la Tabla 4.1.

Con este nuevo marco, el diseño de un WF puede verse como la construcción de un SBC. Es más, como se verá en el Capítulo 6, los WFs especificados en este metamodelo se ejecutarán como los tradicionales SBCs de forma que, además de permitir la típicas funcionalidades aportadas por los WMS, también se podrá razonar sobre los hechos

de la base de conocimiento. Además, el marco propuesto no rompe con el modo tradicional de construir un WF: se mantienen todas las dimensiones de un WF y éstas se modelan en la forma más habitual. Por una parte, la estructura de los WFs se establece a través de redes de Petri, uno de los formalismos más aceptados de diseño de WFs. Por otro lado, los recursos se clasifican dentro del modelo de la organización y se asocian a las actividades que pueden realizar.

En el próximo capítulo prestaremos una especial atención a una de las principales aportaciones de esta tesis doctoral: *el modelo de control*. El modelo de control es el punto en el que se integra el modelado de procesos dentro del metamodelo propuesto. Por ello, se esbozaron las principales características de este modelo basado en redes de Petri jerárquicas haciendo especial énfasis en su anotación algebraica, en su contexto de ejecución y en el modo en el que se construyen estas redes a partir de los patrones de WFs descritos en el Capítulo 2.

5

Modelo de control

Una de las partes esenciales del metamodelo de flujos de trabajo (WFs, del inglés *Workflows*) descrito en esta tesis doctoral es la *dimensión de procesos*. Esta dimensión trata la descripción de problemas de WFs y su resolución a través de métodos sencillos o compuestos. Un componente de esta dimensión es el modelo de control que describe la estructura de los procesos en términos de redes de Petri [266] y que unida a los demás componentes de la dimensión proporciona la descripción operacional de los métodos compuestos, es decir, la forma en la que se coordinan las tareas en las que se descompone un método. En este capítulo se analizarán los detalles más característicos de las redes de Petri que dan soporte a esta parte del metamodelo prestando una especial atención a su estructura y a su anotación algebraica. Asimismo también se mostrará la forma en la que se implementan algunos de los patrones de control más característicos de los WFs en este nuevo contexto.

5.1. Ontologías que soportan el modelo de control

La Figura 5.1 representa la pila de ontologías que da soporte al modelo de control. Se puede observar que este modelo se construye *exclusivamente* en términos de redes de Petri: es una *HLPN jerárquica* que representa un proceso compuesto y que está formada por las redes que están definidas en la capa de patrones de WFs. Esta capa captura los patrones vistos en el Capítulo 2 adaptados al tipo de WFs que se pretenden crear y donde cada comportamiento se representa a través de una HLPN que podrá usarse para componer la red jerárquica. Por ejemplo, existe un patrón para modelar el comportamiento de un proceso y patrones para modelar su estructura interna. Mencionar que estos patrones estructurales son típicos de los WFs [260] y facilitan la estructuración de los procesos. Proporcionan una capa de abstracción que facilita la creación de WFs de forma que los diseñadores de WFs simplemente tendrán que seleccionar el patrón que mejor se adapta a la estructura de su proceso. No tendrán que diseñar dicha estructura a través de una nueva HLPN. Mencionar también que el uso de patrones restringe la expresividad del modelo. Se podrían crear nuevas estructuras directamente a través de HLPNs o incluso estructuras que nada tienen que ver con los WFs. Sin embargo, los diseñadores de WFs están acostumbrados a este

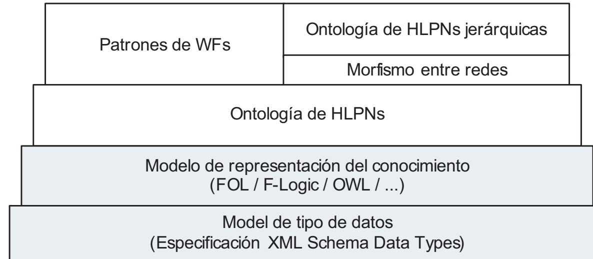


Figura 5.1: Pila de ontologías que da soporte a la representación del modelo de control

tipo de construcciones y, por lo tanto, el uso directo de HLPNs en lugar de patrones dificultaría su trabajo.

5.2. Estructura del modelo de control

El modelo de control tiene un papel fundamental en el metamodelo presentado en esta tesis doctoral, ya que es el punto en el que se integra el modelado de procesos (característico de los WFs) con el modelado del conocimiento (característico de los SBCs). Al más alto nivel, un modelo de control es un proceso modelado a través de redes de Petri. El uso de estas redes para el modelado de WFs no es una novedad: las redes de Petri son el formalismo de representación más empleado tanto para el modelado de procesos de negocio como de aplicaciones industriales. Ello se debe a que son una herramienta que permite abarcar tanto el modelado de sistemas como su verificación e implementación. Las ventajas de modelar WFs mediante redes de Petri son conocidas:

- Aportan un modelado gráfico y matemático. Muy pocas técnicas combinan ambas características, lo cual, facilita su aprendizaje, su uso y también aporta la seguridad de estar soportado por un formalismo matemático.
- Disponen de una gran variedad de algoritmos para el análisis, diseño y simulación. Entre otros aspectos permiten verificar propiedades, detectar ciclos mortales, reconocer modelos o analizar los estados y sus relaciones dentro de una ejecución.
- Permiten modelar explícitamente la concurrencia. Las redes de Petri fueron la primera investigación en modelar formalmente la concurrencia y llevan más de cuarenta años investigando en esta materia.

Sin embargo, el uso de redes de Petri también tiene un importante inconveniente: para modelos complejos la explosión de estados hace que su uso esté desaconsejado; es más, la gran cantidad de plazas, transiciones y arcos que componen una red compleja hace imposible una representación gráfica. En cualquier caso, este inconveniente se suaviza en el caso de las HLPNs.

Los WFs son habitualmente procesos complejos y por ello su representación mediante una HLPN puede estar formada por cientos o incluso miles de nodos. Con esta cantidad de elementos la tarea de diseño se hace casi imposible. Se podría pensar que las HLPNs no están indicadas para esta clase de problemas, sin embargo, si el diseño se parte en piezas más pequeñas, como se hace con las funciones en un lenguaje de programación, la tarea se simplifica. Siguiendo esta aproximación se crearon las redes jerárquicas, que proporcionan una capa de abstracción por encima de las HLPNs, al igual que los lenguajes de programación de tercera generación sobre el lenguaje ensamblador, y que consisten en la unión de varias HLPNs.

En base al razonamiento anterior se decidió representar el modelo de control, concepto `ControlModel`, como una red jerárquica:

```
subClassOf(ControlModel, KnowledgeComponent).
subClassOf(ControlModel, HHLPN).
```

Por lo tanto, un modelo de control estará compuesto por un conjunto de HLPNs denominadas páginas y por un conjunto de mecanismos de composición (*sustituciones* y *fusiones*) que permiten relacionar las *páginas* de la red entre si.

5.2.1. Propiedades que debería cumplir la red jerárquica

La selección de las propiedades que un modelo debe cumplir depende en gran medida de las características del problema a resolver. Sin embargo, es recomendable que cumpla ciertas propiedades y, por ello, en nuestros WFs intentamos que se amolden a las propiedades mencionadas en [255], es decir, un WF debería verificar que:

- Existe una única plaza de *inicio* y una única plaza *final*. Esta propiedad busca que exista un único punto de inicio en el flujo de trabajo y un único punto de finalización, es decir, *(i)* que el flujo no pueda iniciarse en distintos puntos de la red y *(ii)* que se sepa con total seguridad que su ejecución ha finalizado. Esta asunción es razonable, ya que las aplicaciones necesitan saber cómo iniciar un servicio y cuando éste ha finalizado.

- La red es *fuertemente conexa* (*strongly connected* en inglés), es decir, cada transición debe de estar en uno de los caminos que van desde la plaza inicial hasta la plaza final del flujo de trabajo. Un WF que cumpla esta propiedad asegura que cada una de las plazas y transiciones de la red contribuyen al procesamiento del servicio. Profundizando más, esta propiedad contribuye a asegurar que cada uno de los (sub)procesos formará parte de un caso específico a tratar y, por lo tanto, no está desligado de la ejecución.

- La red es *segura* (*safe* en inglés), es decir, sólo puede haber una marca en cada plaza en cualquier estado de la red. Esta propiedad asegura que el WF ejecute un único caso al mismo tiempo.

También es necesario analizar la forma en la que se ejecuta el WF. Para ello suele ser necesario un *análisis de alcanzabilidad* (*reachability analysis* en inglés) para averiguar cuándo y cómo la red puede alcanzar un determinado estado, es decir, las diferentes secuencias de ejecución del procesos:

- La res es *sin bloqueos*. Este análisis busca situaciones donde la ejecución de un proceso compuesto no ha finalizado y no existen transiciones activas. Cuando no se encuentran bloqueos, la red no tiene puntos muertos y, por lo tanto, es posible ejecutar cualquier (sub)proceso siguiendo una ruta determinada.
- La red garantiza la ejecución de un (sub)servicio para un caso determinado. Esta propiedad indica que la red es *viva* (*liveness* en inglés), y asegura que para una marca específica existe una secuencia de ocurrencias que la conducen a un determinado (sub)proceso.
- Cuando la ejecución de la red ha finalizado, la red debe tener una marca en la plaza de salida y todas las demás plazas de la red estar vacías. De esta forma se asegura que todos los (sub)procesos están inactivos al finalizar la ejecución y, por lo tanto, que no queda ninguno pendiente de ejecución.

Para completar el análisis también es necesario comprobar si las redes de Petri resultantes son WFs *bien estructurados* (*well-structured* en inglés) y si pueden clasificarse como redes de Petri *libres de conflictos* (*free-choice* en inglés). La primera de estas propiedades asegura que cada una de las separaciones tiene una unión, es decir, que las separaciones y uniones están balanceadas. La segunda propiedad comprueba que si alguna transición comparte una plaza de entrada p_i, entonces dicha plaza es su única entrada. Por consiguiente, todas las plazas estarán activas o inactivas al mismo tiempo. Por lo tanto, siempre es posible seleccionar libremente cual de ellas vaya a ocurrir sin que su ocurrencia afecte a las demás. Es decir, que la ocurrencia de un servicio no afecte a la ocurrencia de servicios con los que no está relacionado.

5.2.2. Características de la red jerárquica

Aunque existía la posibilidad de no restringir la estructura de la red jerárquica del modelo de control, en esta tesis doctoral se optó por limitar su expresividad. Esta decisión se fundamentó en el concepto de diseño paramétrico que guía la creación de WFs en el metamodelo propuesto en el Capítulo 4 y que aboga por construir un modelo a partir de piezas independientes como si se tratase de un rompecabezas. La limitación impuesta restringe el tipo de HLPNs que pueden formar parte del modelo de control y, por consiguiente, también el tipo de sustituciones y fusiones que se pueden realizar. Por un lado se exige que las páginas sean de uno de los siguientes tipos:

- Patrón proceso (`ProcessPattern`). Se utiliza para definir la HLPN que controla la ejecución del WF.

- Patrón de control (`ControlPattern`). En este tipo de páginas se incluirán las HLPNs que representan los patrones de control de WFs estudiados en el Capítulo 2. En otras palabras, son las HLPNs que coordinan realmente la ejecución de las actividades a realizar en el WF.

Por otro lado, las sustituciones también se limitan a dos tipos:

- Sustitución de la ejecución (`ExecuteSubstitution`). Se aplica a páginas del tipo `ProcessPattern` y permite sustituir la transición que lanza la ejecución del proceso por una red de tipo `ControlPattern`.
- Sustitución del control (`ControlSubstitution`). Permiten sustituir determinadas transiciones de un patrón de control, páginas del tipo `ControlPattern`, por otro patrón de control. A través de este tipo de sustitución se facilita la composición de patrones de control.

Las características anteriores se recogen formalmente en los siguientes axiomas que restringen la definición de un modelo de control:

```
allValuesFrom(hasPages, [ProcessPattern, ControlPattern]).
allValuesFrom(hasSubstitutions, [ExecuteSubstitution, ControlSubstitution]).
```

5.2.3. Raíz de la red jerárquica

Para facilitar el diseño de un WF, el modelo de red jerárquica ha de partir de una HLPN en la que se capturen los componentes esenciales del WF, es decir, la red del WF debe tener un punto donde se comprueben sus precondiciones, un punto donde se permita enganchar su estructura interna, y finalmente otro punto donde se pueda comprobar que cumple con las postcondiciones impuestas. Esta red estará asociada al modelo de control a través de la relación `hasRoot` y su rango será del tipo `ProcessPattern`:

```
objectProperty(hasRoot).

domain(hasRoot, ControlModel).
range(hasRoot, ProcessPattern).
cardinality(hasRoot, 1).
```

Todos los modelos de control diseñados en el metamodelo partirán de redes del tipo `ProcessPattern` a partir de las cuales se iniciará la descomposición de la red jerárquica. Constituirán, por lo tanto, el nodo raíz del árbol de HLPNs que representa la red jerárquica, y por cada modelo de control habrá únicamente un nodo raíz. Además, también se impone la restricción de que exista una única instancia del concepto `ProcessPattern` por cada red jerárquica. Esta última restricción se debe a que cada modelo de control representa una única instancia de proceso y si se creasen dos páginas del tipo `ProcessPattern` significaría que existen (sub)procesos, lo cual carece de

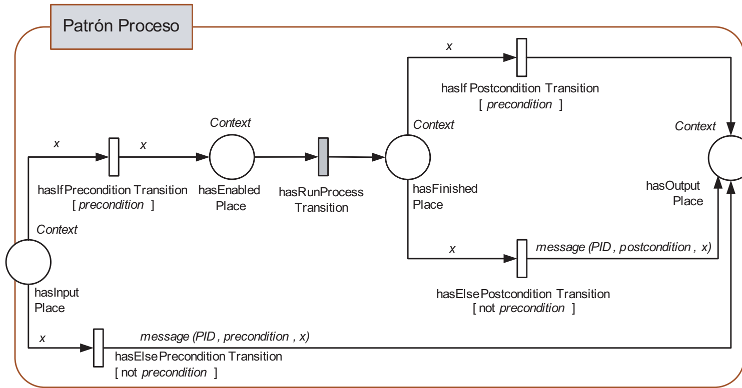


Figura 5.2: Patrón utilizado para la representación de un proceso

sentido en la descripción operacional de un método compuesto. Si bien los métodos compuestos se descomponen en (sub)tareas, hasta que se crea el modelo de ejecución del WF no se sustituyen por los métodos que las implementan. Por lo tanto, la descripción operacional de un método compuesto sólo coordina sus (sub)tareas y ello se resuelve sin la necesidad de tener (sub)procesos.

5.3. Patrón proceso

La HLPN que se muestra en la Figura 5.2 representa el *patrón proceso* y guiará la ejecución de cualquier método compuesto de nuestro metamodelo de WFs. La red tiene dos interfaces: la plaza `hasInputPlace` a la izquierda es la entrada del proceso, mientras que la plaza `hasOutputPlace` a la derecha representa su salida. La estructura representada dentro del marco controla el comportamiento de un proceso al ser ejecutado:

- Un proceso estará activo cuando se cumplan sus precondiciones, es decir, cuando el término *precondition* de la transición `hasIfPreconditionTransition` se verifique.
- La transición `hasRunProcessTransition` se encargará de realizar la ejecución del proceso.
- A su finalización se controlará que el proceso cumple con sus postcondiciones, es decir, si el término *postcondition* de la transición `hasIfPostconditionTransition` se verifica.

Finalmente, la razón por la cual la transición `hasRunProcessTransition` está coloreada en gris es que será sustituida por otra red que modele el control del proceso, es decir, por uno de los patrones de control de WFs.

El concepto ProcessPattern dará soporte a las redes que implementan la estructura de la Figura 5.2, con la que es posible ejecutar cualquier proceso y que constituye la raíz del WF:

```
subClassOf(ProcessPattern, HLPN).
```

Un ProcessPattern se declara como una HLPN y, por lo tanto, hereda los atributos que permiten describir la red como un grafo anotado; es decir, tendrá un conjunto de nodos, arcos y anotaciones, una firma sintáctica y un álgebra. Sin embargo, ya que este concepto modela únicamente instancias del patrón *proceso* es necesario restringir el valor que esos nodos, arcos y anotaciones pueden tomar. Para ello, este concepto identifica cada uno de los nodos de la HLPN representada en la Figura 5.2 y restringe sus características. Además, los arcos, y con ello la estructura de la red, se verificarán a través de los axiomas recogidos en la Tabla 5.1. Las principales características de las relaciones de este concepto son las siguientes:

- Relación hasInputPlace. Permite identificar la interfaz de entrada de este patrón:

  ```
  objectProperty(hasInputPlace).
  domain(hasInputPlace, ProcessPattern).
  range(hasInputPlace, Place).
  cardinality(hasInputPlace, 1).
  ```

  La plaza debe cumplir tres requisitos. El primero es tener como color al contexto de ejecución, es decir, a la clase Context. El segundo es que la firma (y, por lo tanto, el álgebra) contengan los operadores que relacionan el contexto de ejecución con los parámetros de entrada del proceso. El tercer requisito se refiere a la integridad estructural de la plaza, e indica que debe estar con las transiciones hasIfPreconditionTransition y hasElsePreconditionTransition.

- Relación hasOutputPlace. Permite identificar la interfaz de salida de este patrón:

  ```
  objectProperty(hasOutputPlace).
  domain(hasOutputPlace, ProcessPattern).
  range(hasOutputPlace, Place).
  cardinality(hasOutputPlace, 1).
  ```

  Al igual que la plaza de entrada, esta plaza debe estar anotada por el contexto de ejecución y, además, tiene que haber un arco desde las transiciones hasIfPostconditionTransition, hasElsePostconditionTransition y hasElsePreconditionTransition hacia ella.

- Relación hasEnabledPlace. Esta relación se refiere a la entrada del proceso aunque una vez verificada su precondición:

Tabla 5.1: Axiomas que restringen la semántica del patrón *proceso*

Existe un único arco entre la plaza de entrada y la transición `hasIfPreconditionTransition`, y este arco está anotado con una variable.
$\forall R,P,T\ \mathit{ProcessPattern}(R) \land \mathit{hasInputPlace}(R,P) \land \mathit{hasIfPreconditionTransition}(R,T) \rightarrow \mathit{uniqueArc}(P,T) \land \mathit{variableAnnotated}(P,T)$
Existe un único arco entre la plaza de entrada y la transición `hasElsePreconditionTransition`, y este arco está anotado con una variable.
$\forall R,P,T\ \mathit{ProcessPattern}(R) \land \mathit{hasInputPlace}(R,P) \land \mathit{hasElsePreconditionTransition}(R,T) \rightarrow \mathit{uniqueArc}(P,T) \land \mathit{variableAnnotated}(P,T)$
Existe un único arco entre la plaza `hasEnabledPlace` y la transición `hasRunProcessTransition`.
$\forall R,P,T\ \mathit{ProcessPattern}(R) \land \mathit{hasEnabledPlace}(R,P) \land \mathit{hasRunProcessTransition}(R,T) \rightarrow \mathit{uniqueArc}(P,T)$
Existe un único arco entre la plaza `hasFinishedPlace` y la transición `hasIfPostconditionTransition`, y este arco está anotado con una variable.
$\forall R,P,T\ \mathit{ProcessPattern}(R) \land \mathit{hasFinishedPlace}(R,P) \land \mathit{hasIfPostconditionTransition}(R,T) \rightarrow \mathit{uniqueArc}(P,T) \land \mathit{variableAnnotated}(P,T)$
Existe un único arco entre la plaza `hasFinishedPlace` y la transición `hasElsePostconditionTransition`, y este arco está anotado con una variable.
$\forall R,P,T\ \mathit{ProcessPattern}(R) \land \mathit{hasFinishedPlace}(R,P) \land \mathit{hasElsePostconditionTransition}(R,T) \rightarrow \mathit{uniqueArc}(P,T) \land \mathit{variableAnnotated}(P,T)$
Existe un único arco entre la plaza `hasFinishedPlace` y la plaza `hasNoResultTransition`, y este arco está anotado con una variable.
$\forall R,P,T\ \mathit{ProcessPattern}(R) \land \mathit{hasFinishedPlace}(R,P) \land \mathit{hasNoResultTransition}(R,T) \rightarrow \mathit{uniqueArc}(P,T) \land \mathit{variableAnnotated}(P,T)$
Existe un único arco entre la transición `hasIfPreconditionTransition` y la plaza `hasEnabledPlace`, y este arco está anotado con una variable.
$\forall R,P,T\ \mathit{ProcessPattern}(R) \land \mathit{hasEnabledPlace}(R,P) \land \mathit{hasIfPreconditionTransition}(R,T) \rightarrow \mathit{uniqueArc}(T,P) \land \mathit{variableAnnotated}(T,P)$
Existe un único arco entre la transición `hasElsePreconditionTransition` y la plaza de salida, y este arco está anotado con una variable.
$\forall R,P,T\ \mathit{ProcessPattern}(R) \land \mathit{hasOutputPlace}(R,P) \land \mathit{hasElsePreconditionTransition}(R,T) \rightarrow \mathit{uniqueArc}(T,P) \land \mathit{variableAnnotated}(T,P)$
Existe un único arco entre la transición `hasRunProcessTransition` y la plaza `hasFinishedPlace`.
$\forall R,P,T\ \mathit{ProcessPattern}(R) \land \mathit{hasFinishedPlace}(R,P) \land \mathit{hasRunProcessTransition}(R,T) \rightarrow \mathit{uniqueArc}(T,P)$
Existe un único arco entre la transición `hasIfPostconditionTransition` y la plaza `hasEnabledPlace`, y este arco está anotado con una variable.
$\forall R,P,T\ \mathit{ProcessPattern}(R) \land \mathit{hasEnabledPlace}(R,P) \land \mathit{hasIfPostconditionTransition}(R,T) \rightarrow \mathit{uniqueArc}(T,P) \land \mathit{variableAnnotated}(T,P)$
Existe un único arco entre la transición `hasElsePostconditionTransition` y la plaza de salida, y este arco está anotado con una variable.
$\forall R,P,T\ \mathit{ProcessPattern}(R) \land \mathit{hasOutputPlace}(R,P) \land \mathit{hasElsePostconditionTransition}(R,T) \rightarrow \mathit{uniqueArc}(T,P) \land \mathit{variableAnnotated}(T,P)$

```
objectProperty(hasEnabledPlace).
domain(hasEnabledPlace, ProcessPattern).
range(hasEnabledPlace, Place).
cardinality(hasEnabledPlace, 1).
```

Por lo tanto, las marcas incluidas en esta plaza se utilizarán como entradas de la transición `hasRunProcessTransition` y, consecuentemente, del núcleo encargado de la ejecución del proceso. Esta plaza debe cumplir las mismas restricciones que la plaza de entrada del patrón respecto a los datos que debe contener el contexto. En cuanto a la estructura, únicamente ha de verificar que está conectada con la transición `hasRunProcessTransition` y que tanto la plaza como el arco que salen de la misma están anotados adecuadamente.

- Relación `hasFinishedPlace`. Se refiere a la plaza que contiene la salida del núcleo encargado de la ejecución del proceso:

```
objectProperty(hasFinishedPlace).
domain(hasFinishedPlace, ProcessPattern).
range(hasFinishedPlace, Place).
cardinality(hasFinishedPlace, 1).
```

Esta plaza almacenará temporalmente la salida de la transición `hasRunProcessTransition` que se usará para verificar que la postcondición del proceso se ha cumplido. Por ello, esta plaza tiene un arco hacia las transiciones `hasIfPostconditionTransition`y `hasElsePostConditionTransition`.

- Relación `hasIfPreconditionTransition`. Permite identificar la transición que actúa como precondición del proceso:

```
objectProperty(hasIfPreconditionTransition).
domain(hasIfPreconditionTransition, ProcessPattern).
range(hasIfPreconditionTransition, Transition).
cardinality(hasIfPreconditionTransition, 1).
```

La condición de guarda de la transición `hasIfPreconditionTransition` se anotará con la precondición del proceso, y estará activa cuando haya un contexto de ejecución que verifique esa condición. En el caso de que el proceso no tenga precondición, la transición `hasIfPreconditionTransition` se anotará con una llamada al operador *true* y, por lo tanto, siempre se verificará.

- Relación `hasElsePreconditionTransition`. El patrón *proceso* introduce una rama alternativa para tratar los casos en los que la precondición del proceso no se verifica. Cuando la transición `hasElsePreconditionTransition` se activa, la precondición del proceso no se verifica y, por lo tanto, éste no se ejecuta. Comentar que también existe una razón funcional para añadir esta rama a la red: mejorar su diseño y que de esta forma la red sea segura y libre de ciclos mortales. La siguiente relación permite identificar esta transición:

```
objectProperty(hasElsePreconditionTransition).
domain(hasElsePreconditionTransition, ProcessPattern).
range(hasElsePreconditionTransition, Transition).
cardinality(hasElsePreconditionTransition, 1).
```

Además, el arco de salida de esta transición se anota con un operador cuyo propósito es indicar que las entradas no verifican la precondición del proceso. Finalmente, comentar que la condición de guardia de la transición se anota con la precondición negada del proceso.

- Relación `hasRunProcessTransition`. Permite identificar la transición que modela la ejecución del proceso:

```
objectProperty(hasRunProcessTransition).
domain(hasRunProcessTransition, ProcessPattern).
range(hasRunProcessTransition, Transition).
cardinality(hasRunProcessTransition, 1).
```

  Como se comentó anteriormente, esta transición será sustituida por una red más compleja que contendrá la estructura del WF. Por ello, esta transición únicamente debe cumplir los requisitos estructurales del patrón *proceso*.

- Relación `hasIfPostconditionTransition`. Identifica la transición que actúa como precondición del proceso:

```
objectProperty(hasIfPostconditionTransition).
domain(hasIfPostconditionTransition, ProcessPattern).
range(hasIfPostconditionTransition, Transition).
cardinality(hasIfPostconditionTransition, 1).
```

  La condición de guarda de la transición `hasIfPreconditionTransition` se anotará con la postcondición del proceso, y estará activa cuando haya un contexto de ejecución que verifique esa condición. Si no existen postcondiciones, la transición `hasIfPostconditionTransition` se anotará con una llamada al operador *true* y, por lo tanto, siempre se verificará.

- Relación `hasElsePostconditionTransition`. La transición `hasElsePostconditionTransition` trata los casos en los que la postcondición del proceso no se verifica. La siguiente relación permite identificar esta transición:

```
objectProperty(hasElsePostconditionTransition).
domain(hasElsePostconditionTransition, ProcessPattern).
range(hasElsePostconditionTransition, Transition).
cardinality(hasElsePostconditionTransition, 1).
```

  Esta transición tiene por condición de guardia a la postcondición del proceso negada y anota su arco de salida con el operador *message* para así indicar que no se cumple la precondición.

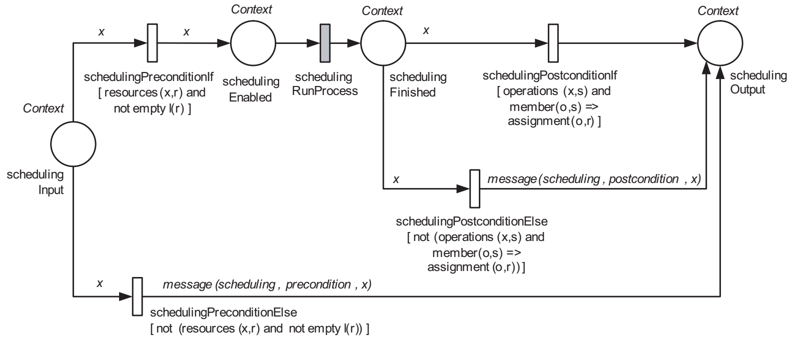


Figura 5.3: Ejemplo de aplicación del patrón *proceso* para un proceso de asignación de recursos

El patrón representado en la Figura 5.2 debe reescribirse adecuándose a las características de cada proceso. Esto significa adaptar la anotación de la red, como se muestra en la Figura 5.3 para un proceso de asignación de recursos a un conjunto de procesos de fabricación. En este caso, las condiciones de guardia de las transiciones que controlan las pre/postcondiciones están anotadas con sus correspondientes fórmulas. Finalmente, es interesante remarcar que todas las plazas están anotadas `Context` y no con un color específico para el proceso. Ello se debe a que la particularización del contexto de ejecución para cada proceso se realiza a nivel del álgebra, es decir, a través de los puentes que relacionan el modelo de control con los modelos del dominio y con los métodos.

5.4. Patrones de flujos de trabajo

El modelo de control especifica a través de una HLPN jerárquica la estructura que coordinará la ejecución de las (sub)tareas que componen un método compuesto. Como se comentó en los apartados anteriores, todos los procesos parten del *patrón proceso*, es decir, su ejecución estará guiada por la HLPN representada en la Figura 5.2. Tomando como base esta red surge la pregunta de ¿cómo se diferencian los procesos entre sí y cómo se establece su estructura interna? Simplemente sustituyendo transición `hasRunProcessTransition`: esta transición representa una estructura *abstracta* y por ello al concretar un WF es imprescindible detallarla. En el caso de las redes jerárquicas, la especificación de un nodo abstracto se realiza mediante la sustitución del nodo por una red que implemente la estructura deseada. Cabe mencionar que la red sustituta puede a su vez tener nodos abstractos que han de sustituirse nuevamente por otra red. En tal caso y para distinguirlos de los demás nodos, las representaciones gráficas de la red los colorearán en gris. En cualquier caso, tampoco se admite cualquier tipo de sustitución para fijar la estructura interna del proceso: todas las redes sustitutas han de ser del tipo `ControlPattern` (acrónimo del inglés *Workflow Pattern*). Algunos de los patrones aparecen en la jerarquía representada en la Figura

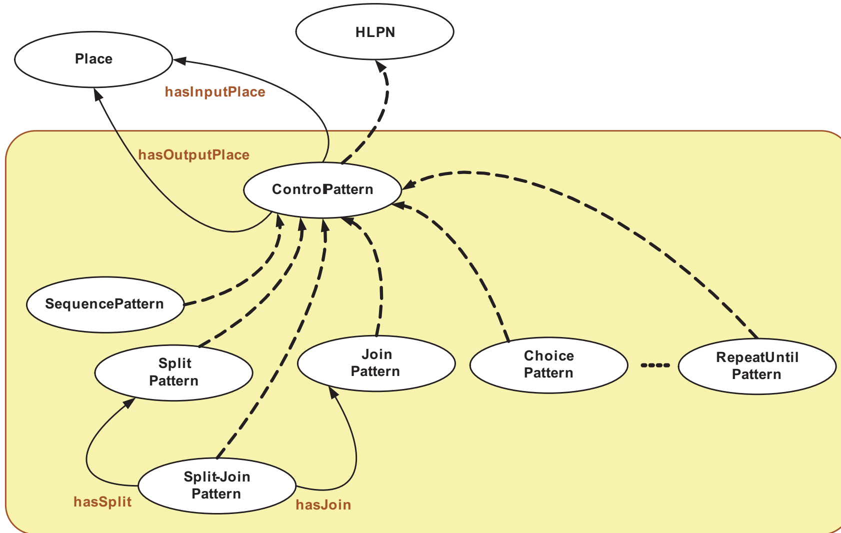


Figura 5.4: Red semántica de la taxonomía de patrones de flujos de trabajo

5.4 y como es fácil comprobar se corresponden con los patrones de WFs vistos en el Capítulo 2:

```
subClassOf(ControlPattern, HLPN).

domain(hasInputPlace, ControlPattern).
domain(hasOutputPlace, ControlPattern).
```

Este concepto establece que un patrón de WFs tendrá siempre una única plaza de entrada y una única plaza de salida referidas a través de las relaciones `hasInputPlace` y `hasOutputPlace`, respectivamente. Esta condición es imprescindible para asegurar la interpretabilidad del WF, ya que con varios puntos de entrada o de salida sería complicado saber cuándo el WF ha comenzado o cuándo ha finalizado.

A continuación se detallarán las HLPNs y los conceptos que representan algunos de los patrones de comportamiento descritos en el Capítulo 2. Es necesario mencionar que no es un objetivo de esta tesis modelar todos los patrones de WFs y por ello en la siguiente sección únicamente incluimos aquellos que son más representativos.

5.4.0.1. Patrón secuencia (control básico)

Este patrón modela la ejecución ordenada de un conjunto de procesos. Aunque las redes de Petri no disponen de un mecanismo específico para la creación de secuencias, éstas se definen en función de las dependencias/arcos del grafo. La Figura 5.5 representa el patrón *secuencia* para dos construcciones de control. En este caso, la

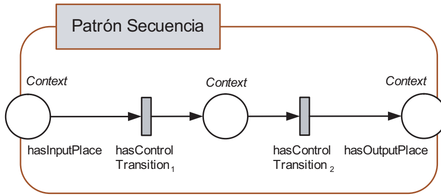


Figura 5.5: Representación del patrón secuencia

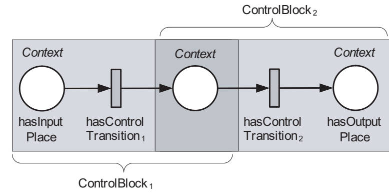


Figura 5.6: Bloques de control del patrón secuencia

transición `hasControlTransition`$_2$ no podrá iniciarse hasta que la transición `hasControlTransition`$_1$ haya finalizado su ejecución. El concepto `SequencePattern` se refiere a este patrón:

```
subClassOf(SequencePattern, ControlPattern).

objectProperty(hasControlList).
domain(hasControlList, SequencePattern).
range(hasControlList, ControlBlockList).
cardinality(hasControlList, 1).
```

Además de la plaza de entrada y salida, es necesario identificar las plazas intermedias y las transiciones de control que completan la estructura de este patrón. Para ello se introduce la relación `hasControlList` que apunta a una lista de instancias del tipo `ControlBlock`. Cada una de estas instancias representa una red con una plaza de entrada, una transición y una plaza de salida, y donde la plaza de entrada se conecta a la transición y la transición a la plaza de salida. La conexión de estos elementos está reflejada axiomáticamente en la Tabla 5.2.

La Figura 5.6 muestra gráficamente cómo una secuencia de dos procesos necesita de dos bloques de control y cómo en este caso los dos bloques comparten la plaza intermedia. Como puede verse en la siguiente definición, un bloque de control se compone exclusivamente de los tres nodos anteriormente mencionados (entrada, salida y transición) y es una red comodín que se utilizará en muchos de los patrones del modelo:

```
subClassOf(ControlBlock, HLPN).

domain(hasInputPlace, ControlBlock).
domain(hasOutputPlace, ControlBlock).

objectProperty(hasControlTransition).
domain(hasControlTransition, ControlBlock).
range(hasControlTransition, Transition).
cardinality(hasControlTransition, 1).
```

A partir de esta lista de bloques de control y de los axiomas descritos en la Tabla 5.3 se restringe la estructura y características del patrón *secuencia* de la Figura 5.5.

Tabla 5.2: Axiomas que restringen la estructura de una rama de control

Existe un único arco entre la plaza de entrada y la transición de control.
$\forall R, P, T\ ControlBlock(R) \wedge hasInputPlace(R, P) \wedge hasControlTransition(R, T) \rightarrow uniqueArc(P, T)$
Existe un único arco entre la transición de control y la plaza de salida.
$\forall R, P, T\ ControlBlock(R) \wedge hasOutputPlace(R, P) \wedge hasControlTransition(R, T) \rightarrow uniqueArc(T, P)$

Tabla 5.3: Axiomas que restringen la semántica del patrón *secuencia*

Si A es la lista de ramas de control, $\lvert A\rvert > 0$.
$\forall S, L, N\ SequencePattern(S) \wedge hasControlList(S, L) \wedge length(L, N) \rightarrow N > 0$
Si A es la lista de ramas de control, la plaza de salida de la rama i es la plaza de entrada de la rama $i + 1$, $0 < i < \lvert A\rvert$.
$\forall S, P, L, B_1, B_2\ SequencePattern(S) \wedge hasControlList(S, L) \wedge ith(I, L, B_1) \wedge ith(I + 1, L, B_2) \wedge hasOutputPlace(B_1, P) \rightarrow hasInputPlace(B_2, P)$
La plaza de entrada del patrón *secuencia* es la plaza de entrada de la primera rama de control.
$\forall S, P, L, B\ SequencePattern(S) \wedge hasControlList(S, L) \wedge hasInputPlace(S, P) \wedge ith(1, L, B) \rightarrow hasInputPlace(B, P)$
La plaza de salida del patrón *secuencia* es la plaza de salida de la última rama de control.
$\forall S, P, L, B\ SequencePattern(S) \wedge hasControlList(S, L) \wedge hasOutputPlace(S, P) \wedge length(L, N) \wedge ith(N, L, B) \rightarrow hasOutputPlace(B, P)$

Por ejemplo, estos axiomas establecen que la plaza de entrada de la secuencia es la plaza de entrada del primer bloque de control y que la plaza de salida del último bloque de control es la plaza de salida del patrón.

5.4.0.2. Patrones separación y sincronización (control básico)

Algunos de los patrones de WFs descritos en el Capítulo 2 pueden combinarse entre sí para dar lugar a un nuevo comportamiento. Este es el caso de los patrones *separación* y *sincronización*, que combinados son una poderosa e imprescindible de diseño:

- Una *separación* describe una estructura donde un conjunto de procesos pueden ejecutarse concurrentemente. Por lo tanto, a partir de un hilo de ejecución de entrada se lanzan o generan varios hilos de ejecución concurrentes. Por ejemplo, la reserva de un viaje puede lanzar en paralelo tres procesos independientes para la reserva del billete de avión, del hotel y de una oferta turística. El principal inconveniente de este patrón es que no permite responder a la pregunta ¿cuándo finaliza el proceso? La respuesta más habitual sería al finalizar todas las ramas, pero incluso esta opción es discutible, ya que habría que decidir si la finalización ha de ser síncrona o asíncrona. Por este motivo, como se mostrará más adelante, una separación suele combinarse con otro patrón que indique cómo finalizar las ramas paralelas.

Tabla 5.4: Axiomas que restringen la semántica del patrón *separación*

Existe un único arco entre la plaza de entrada y la transición `hasSplitTransition`, y este arco está anotado con una variable.
$\forall R, P, T\ SplitPattern(R) \wedge hasInputPlace(R, P) \wedge hasSplitTransition(R, T) \rightarrow uniqueArc(P, T) \wedge variableAnnotated(P, T)$
Existe un único arco entre la transición `hasSplitTransition` y las plazas de salida, y este arco está anotado con una variable.
$\forall R, P, T\ SplitPattern(R) \wedge hasOutputPlaces(R, P) \wedge hasSplitTransition(R, T) \rightarrow uniqueArc(T, P) \wedge variableAnnotated(T, P)$

El paralelismo se expresa fácilmente mediante una HLPN: ya que el grafo establece las dependencias entre las transiciones a ejecutar, aquellas que no estén en el mismo camino podrán ejecutarse concurrentemente. La Figura 5.7 representa la semántica operacional de una *separación*, que está descrita a través del concepto `SplitPattern`:

```
subClassOf(SplitPattern, ControlPattern).

domain(hasOutputPlaces, SplitPattern).
range(hasOutputPlaces, Place).
```

Una *separación* tiene una plaza de entrada y varias de salida referidas a través de las relaciones `hasInputPlace` y `hasOutputPlaces`, respectivamente. Una parte importante de esta red es la transición `hasSplitTransition` cuyo disparo generará una marca en cada una de las plazas de salida:

```
objectProperty(hasSplitTransition).
domain(hasSplitTransition, SplitPattern).
range(hasSplitTransition, Transition).
cardinality(hasSplitTransition, 1).
```

- Una *sincronización* señala el punto del WF en el que un conjunto de ramas deben unirse. Este patrón indica, por lo tanto, que un conjunto de hilos de ejecución referentes al mismo caso han de sincronizarse para seguir o finalizar el WF. Por ejemplo, antes de pagar un viaje es imprescindible asegurar que el billete de avión, el alojamiento en hotel y la oferta turística se han reservado correctamente. Como su propio nombre indica, la precondición de este patrón es que exista *algo* que sincronizar, y por ello suele usarse en combinación con un patrón que lance varios hilos de ejecución. Además, su uso por separado implica la necesidad de establecer el momento en el que empieza la ejecución de este patrón: ¿comienza cuando alguna de las ramas concurrentes está activa o cuando todas están activas y listas para sincronizarse?

  Las redes de Petri soportan el concepto de *sincronización* a través de su regla de disparo de transiciones: una transición no se puede ejecutar si sus plazas de entrada no tienen las marcas necesarias para verificar *(i)* la evaluación de los arcos

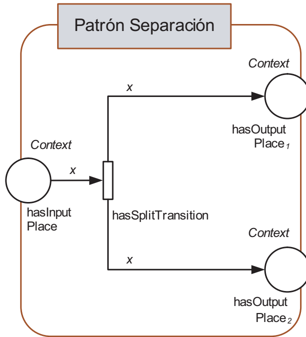


Figura 5.7: Representación del patrón separación

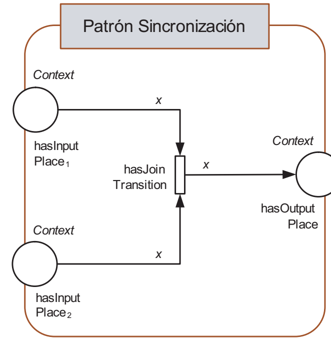


Figura 5.8: Representación del patrón sincronización

de entrada de la transición y *(ii)* la evaluación afirmativa de la precondición de la transición. La Figura 5.8 muestra esta estructura de control donde todas las ramas se unen en la transición `hasJoinTransition`. El concepto `JoinPattern` se refiere a esta red:

```
subClassOf(JoinPattern, ControlPattern).

domain(hasInputPlaces, JoinPattern).
range(hasInputPlaces, Place).
```

La transición `hasJoinTransition` juega un papel fundamental en este patrón, ya que sólo se activará cuando existan las suficientes marcas en las plazas de entrada para verificar su precondición:

```
objectProperty(hasJoinTransition).
domain(hasJoinTransition, JoinPattern).
range(hasJoinTransition, Transition).
cardinality(hasJoinTransition, 1).
```

La ejecución de esta transición eliminará las marcas de las plazas de entrada y generará una nueva marca en la plaza `hasOutputPlace` que identifica el punto de sincronización. La Tabla 5.5 contiene los axiomas que modelan la estructura de la red.

La combinación de los comportamientos *separación* y *sincronización* permite la ejecución concurrente de un conjunto de procesos entre dos puntos de control. Por un lado la *separación* establece el punto a partir del cual un conjunto de procesos pueden ejecutarse concurrentemente. Por el otro, la *sincronización* indica dónde esos procesos deben unirse. Para facilitar el uso de patrones *separación* y *sincronización* en una misma estructura de control se ha creado el concepto `SplitJoinPattern`:

Tabla 5.5: Axiomas que restringen la semántica del patrón *sincronización*

Existe un único arco entre cada plaza de entrada y la transición `hasJoinTransition`, y este arco está anotado con una variable.
$\forall R, P, T\ JoinPattern(R) \wedge hasInputPlaces(R, P) \wedge hasJoinTransition(R, T) \rightarrow uniqueArc(P, T) \wedge variableAnnotated(P, T)$
Existe un único arco entre la transición `hasJoinTransition` y la plaza de salida, y este arco está anotado con una variable.
$\forall R, P, T\ JoinPattern(R) \wedge hasOutputPlace(R, P) \wedge hasJoinTransition(R, T) \rightarrow uniqueArc(T, P) \wedge variableAnnotated(T, P)$

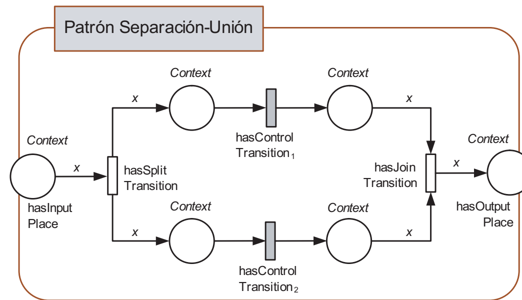


Figura 5.9: Separación y unión de dos hilos de ejecución

```
subClassOf(SplitJoinPattern, ControlPattern).

domain(hasSplit, SplitJoinPattern).
range(hasSplit, SplitPattern).
cardinality(hasSplit, 1).

domain(hasJoin, SplitJoinPattern).
range(hasJoin, JoinPattern).
cardinality(hasJoin, 1).

domain(hasControlBlocks, SplitJoinPattern).
```

La Figura 5.9 representa el patrón *separación-sincronización* en el que pueden distinguirse tres partes:

- La relación `hasSplit` identifica la *separación*.
- La relación `hasJoin` identifica la *sincronización*.
- La relación `hasControlBlocks` identifica las redes de control que unen los puntos de separación con los de unión. Cada una de las ramas de control se modela a través de una instancia del concepto `ControlBlock`, y en la Figura 5.10 se puede ver gráficamente cómo establece las correspondencias entre las separaciones y las sincronizaciones.

Tabla 5.6: Axiomas que restringen la semántica del patrón *separación-unión*

Existen tantas separaciones como uniones.
$\forall R, P, T, N_1, N_2\ SplitJoinPattern(R) \wedge hasSplit(R, P) \wedge hasJoin(R, T) \wedge cardinality(S, hasInputPlaces, N_1) \wedge cardinality(T, hasOutputPlaces, N_2) \rightarrow N_1 = N_2$
Existen tantas separaciones como ramas de control.
$\forall R, P, T, N_1, N_2\ SplitJoinPattern(R) \wedge hasSplit(R, P) \wedge cardinality(S, hasInputPlaces, N_1) \wedge cardinality(R, hasControlBlocks, N_2) \rightarrow N_1 = N_2$
La plaza de entrada de cada bloque de control es una de las plazas de salida de la separación.
$\forall R, T, P\ SplitJoinPattern(R) \wedge hasSplit(T) \wedge hasOutputPlaces(T, P) \rightarrow \exists B\ hasControlBlocks(R, B) \wedge hasInputPlace(B, P)$
La plaza de salida de cada bloque de control es una de las plazas de entrada de la red de sincronización.
$\forall R, T, P\ SplitJoinPattern(R) \wedge hasJoin(T) \wedge hasInputPlaces(T, P) \rightarrow \exists B\ hasControlBlocks(R, B) \wedge hasOutputPlace(B, P)$

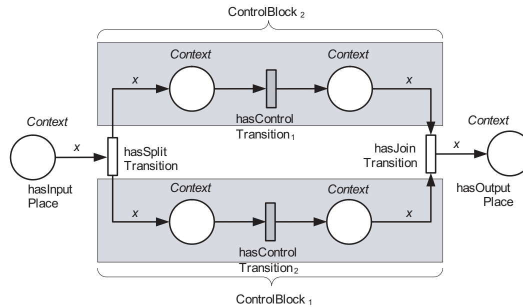


Figura 5.10: Bloques de control del patrón separación-unión

Los axiomas de la Tabla 5.6 completan la definición del patrón *separación-unión*.

5.4.0.3. Patrones elección exclusiva y mezcla simple (control básico)

Los dos últimos patrones de control básico de WFs también suelen utilizarse de forma combinada, ya que complementan sus comportamientos: por un lado, la *elección exclusiva* activa un único hilo de ejecución de un conjunto de hilos paralelos; por el otro, la *mezcla simple* está pensada para un único hilo de los hilos paralelos que convergen se ejecute por cada caso. El comportamiento detallado de estos dos patrones es el siguiente:

- La *elección exclusiva* selecciona una única rama o hilo de ejecución de entre un conjunto de ramas seleccionables. En un WF orientado al dominio educativo, un ejemplo de este comportamiento sería la selección de la asignatura a la

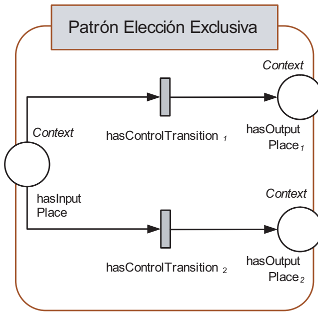


Figura 5.11: Representación del patrón elección

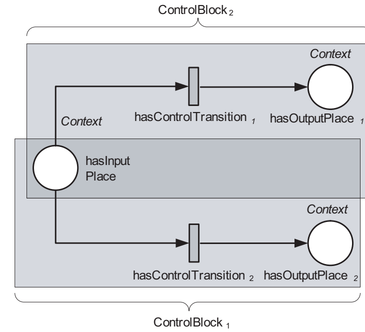


Figura 5.12: Bloques de control del patrón elección

que un usuario quiere asistir de entre el conjunto de asignaturas en las que está matriculado. En el contexto de las redes de Petri el comportamiento de esta estructura de control implica la creación de un punto de conflicto entre varias transiciones. En la Figura 5.11 se representa una solución basada en HLPNs a este patrón: las transiciones compiten por las marcas de la plaza `hasInputPlace` y la selección de uno de los arcos estará a cargo del planificador de tareas del motor de redes de Petri. Se trata, por lo tanto, de una elección no determinista. Existen otras alternativas a esta red donde la elección se basa en criterios definidos en tiempo de ejecución. Por ejemplo, si cada una de las ramas que van de la plaza `hasInputPlace` a las transiciones en conflicto se anota con expresiones mutuamente excluyentes.

El concepto `ChoicePattern` se refiere a este patrón:

```
subClassOf(ChoicePattern, ControlPattern).
domain(hasControlBlocks, ChoicePattern).
```

La red dispone de una plaza de entrada identificada por la relación `hasInputPlace`, que actúa como punto de conflicto, y un conjunto de plazas de salida identificadas a través de la relación `hasOutputPlaces`. La Figura 5.11 también muestra cómo las transiciones `hasControlTransition`$_i$ están en conflicto, de modo que, aunque todas ellas puedan estar activas al mismo tiempo, únicamente una podrá ejecutarse.

La relación `hasControlBlocks` se refiere a cada una de las ramas del patrón. La Figura 5.12 representa gráficamente qué elementos de la HLPN forman parte de ellas. En este caso puede apreciarse cómo la plaza de entrada es compartida por *todas* las ramas del patrón, que, por lo tanto, están en conflicto, permitiendo únicamente la ejecución de una de ellas. En la Tabla 5.7 se muestra el axioma que asegura que la plaza de entrada es común a todos los bloques de control.

- La *mezcla simple* proporciona una forma de combinar, *sin* sincronización, dos o más ramas, de forma que se unifique el comportamiento común de los distintos

Tabla 5.7: Axiomas que restringen la semántica del patrón *elección*

La plaza de entrada del patrón es la misma que la plaza de entrada de cada una de las ramas de control.
$\forall S, B, P\ ChoicePattern(S) \wedge hasControlBlocks(S, B) \wedge hasInputPlace(S, P) \rightarrow hasInputPlace(B, P)$

Tabla 5.8: Axiomas que restringen la semántica del patrón *mezcla simple*

La plaza `hasOutputPlace` es *segura*.
$\forall S, B, P\ MergePattern(S) \wedge hasOutputPlace(S, P) \rightarrow maxBbounded(P, 1)$

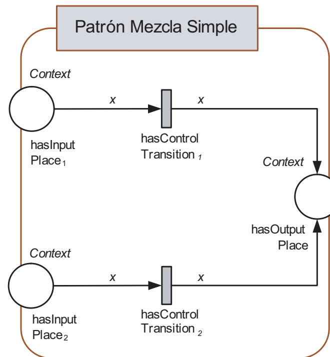


Figura 5.13: Representación de una mezcla simple de dos ramas

hilos de ejecución. Por ejemplo, un cajero automático entrega un recibo al cliente después de retirar el dinero de la cuenta, de recargar el móvil o simplemente de actualizar la libreta de ahorros. La Figura 5.13 muestra el modo en que las redes de Petri soportan este comportamiento: la unión de todas las ramas en una plaza, en este caso `hasOutputPlace`, indica qie la ejecución de cualquiera de las transiciones $\texttt{hasControlTransition}_i$ generará una nueva marca en la plaza `hasOutputPlace`.

El concepto `MergePattern` se refiere a esta red:

```
subClassOf(MergePattern, MultiMergePattern).
```

Como se puede apreciar en la definición anterior, la *mezcla simple* extiende la clase `MultiMergePattern`, que se describirá en el apartado 5.4.0.4, añadiendo una restricción: la plaza en la que se produce la mezcla (por ejemplo, `hasOutputPlace` en la Figura 5.13) es *segura* (*safe*, en inglés) y, por lo tanto, nunca podrá contener más de una marca para el mismo caso de ejecución. Si el modelo no puede asegurar esta restricción, debería usarse la *mezcla múltiple* en lugar de la *mezcla simple*. La Tabla 5.8 muestra este nuevo axioma.

Tabla 5.9: Axiomas que restringen la semántica de la *elección exclusiva-mezcla simple*

La plaza de entrada de la red de *elección* es la misma que la plaza de entrada del patrón.
$\forall S, C, P\ ChoiceMergePattern(S) \wedge hasChoice(S, C) \wedge hasInputPlace(C, P) \rightarrow hasInputPlace(S, P)$
La plaza de salida de la red de *mezcla* es la misma que la plaza de salida del patrón.
$\forall S, M, P\ ChoiceMergePattern(S) \wedge hasMerge(S, M) \wedge hasOutputPlace(M, P) \rightarrow hasOutputPlace(S, P)$
Existe el mismo número de ramas de control en la *elección* y en la *mezcla*.
$\forall S, C, M, N_1, N_2\ ChoiceMergePattern(S) \wedge hasChoice(S, C) \wedge hasMerge(S, M) \wedge cardinality(C, hasControlBlocks, N_1) \wedge cardinality(M, hasControlBlocks, N_2) \rightarrow N_1 = N_2$
La plaza de salida cada bloque de control de la *elección* será plaza de entrada de uno de los bloques de la *mezcla*.
$\forall S, C, M, N_1, N_2\ ChoiceMergePattern(S) \wedge hasChoice(S, C) \wedge hasControlBlocks(C, B_1) \wedge hasOutputPlaces(C, P) \rightarrow \exists M\ hasMerge(S, M) \wedge hasInputPlaces(M, P)$

Precisamente la restricción que limita el número de marcas que puede contener la plaza `hasOutputPlace` hace que la *mezcla simple* suela combinarse con la *elección exclusiva*. Esta combinación está representada en la Figura 5.14 para una red de dos ramas, donde se muestra cósmo las salidas de la elección se unen con las entradas de la mezcla. El concepto `ChoiceMergePattern` hace referencia a este patrón:

```
subClassOf(ChoiceMergePattern, ControlPattern).

domain(hasChoice, ChoiceMergePattern).
range(hasChoice, ChoicePattern).
cardinality(hasChoice, 1).

domain(hasMerge, ChoiceMergePattern).
range(hasMerge, MergePattern).
cardinality(hasMerge, 1).
```

La Tabla 5.9 contiene los axiomas que complementan la semántica de esta red. Por una parte, comprueban que la plaza de entrada del patrón y de la *elección* es la misma, y que la plaza de salida del patrón y de la mezcla también es la misma. Por otra parte, comprueban que cada uno de los bloques de control de la elección encaja con un bloque de control de la mezcla.

5.4.0.4. Patrones elección múltiple y mezcla múltiple (control avanzado)

En este apartado se presentan dos de los patrones de control avanzado que suelen soportar la mayoría de los sistemas de WF comerciales (Tabla 5.10): la *elección múltiple* y la *mezcla múltiple*, que tienen un comportamiento complementario:

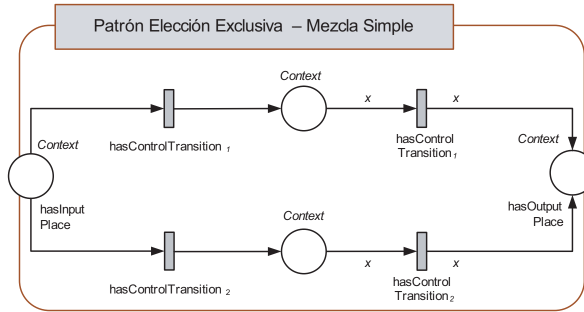


Figura 5.14: Representación del patrón elección exclusiva-mezcla simpe para dos ramas

Tabla 5.10: Patrones de control avanzado soportados por los principales sistemas de WFs comerciales [260] (1-Staffware, 2-WebSphere MQ Workflow, 3-FLOWer, 4-COSA, 5-iPlanet, 6-SAP Workflow, 7-FileNet). Los signos '+' indican un soporte directo, los '+/-' un soporte parcial y los '-' que el lenguaje no da soporte al patrón. Un '-' no implica que sea imposible resolver el patrón a través de un rodeo: implica que la solución no es directa.

Nombre del patrón	1	2	3	4	5	6	7
Control avanzado							
Elección múltiple	-	+	+	+	+	-	+
Mezcla múltiple	-	-	+/-	-	+	-	+
Mezcla sincronizada estr.	-	+	+	-	-	-	+
Discriminador estructurado	-	-	-	-	+	+/-	-

- La *elección múltiple*, al contrario que la elección exclusiva, posibilita la selección de más de una de las ramas de control. Por ejemplo, este patrón podría usarse para automatizar un WF que en función de las características de una llamada de emergencia contacte con la policía, los bomberos, una ambulancia o varios a la vez. La Figura 5.15 muestra la estructura de este patrón: usando varios criterios de selección el hilo de control de la red se divide en varios hilos concurrentes. La estructura es idéntica a la red de *separación*, aunque se diferencia en la anotación de los arcos de salida de la transición `hasSplitTransition`. Estas anotaciones se evaluarán en tiempo de ejecución y en función del resultado se activarán ninguno, todos o alguno de los hilos de ejecución de las ramas concurrentes. El concepto `MultiMergePattern` se refiere a este comportamiento:

```
subClassOf(MultiChoicePattern, ControlPattern).
domain(hasOutputPlaces, MultiChoicePattern).
```

  Una *elección múltiple* tiene una plaza de entrada y varias de salida referidas a través de las relaciones `hasInputPlace` y `hasOutputPlaces`, respectivamente. La transición `hasSplitTransition` se utiliza para unir la plaza de entrada con cada una de las plazas de salida:

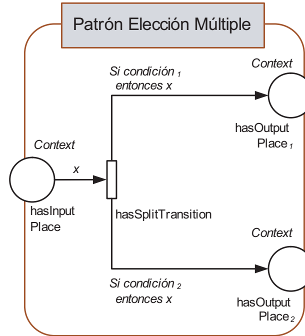


Figura 5.15: Representación de la elección múltiple de dos ramas

Tabla 5.11: Axiomas que restringen el patrón *elección múltiple*

Existe un único arco entre la plaza de entrada y la transición `hasSplitTransition`, y este arco está anotado con una variable.
$\forall R, P, T\ MultiChoicePattern(R) \wedge hasInputPlace(R, P) \wedge hasSplitTransition(R, T) \rightarrow uniqueArc(P, T) \wedge variableAnnotated(P, T)$
Existe un único arco entre la transición `hasSplitTransition` y las plazas de salida, y este arco está anotado con una expresión.
$\forall R, P, T\ MultiChoicePattern(R) \wedge hasOutputPlaces(R, P) \wedge hasSplitTransition(R, T) \rightarrow uniqueArc(T, P) \wedge expressionAnnotated(T, P)$

```
objectProperty(hasSplitTransition).
domain(hasSplitTransition, SplitPattern).
range(hasSplitTransition, Transition).
cardinality(hasSplitTransition, 1).
```

La Tabla 5.11 complementa la definición anterior con los axiomas que verifican la estructura y anotación de la red.

- La *mezcla múltiple* facilita la convergencia de dos o más ramas en un único hilo de ejecución. La Figura 5.16 muestra la estructura de este patrón que, como se puede apreciar, es idéntica a la propuesta para la mezcla simple. La diferencia radica en que la mezcla múltiple no limita el número de ramas activas para una misma instancia y, por lo tanto, la plaza `hasOutputPlace` pueda contener varias marcas referidas al mismo caso (es decir, que no sea *segura*). Este tipo de comportamiento suele encontrarse en WFs del mundo real, como por ejemplo, en un procedimiento de presupuestado en el que la estimación del coste de material y la estimación del coste de fabricación ocurra en paralelo, pero donde la revisión de ambos procesos converja en un mismo hilo de ejecución en tiempos distintos. El concepto `MultiMergePattern` se refiere a esta red:

```
subClassOf(MultiMergePattern, ControlPattern).
domain(hasControlBlocks, MultiMergePattern).
```

Tabla 5.12: Axiomas que restringen la semántica del patrón *mezcla múltiple*

Las ramas de control comparten la plaza de salida.
$\forall S, B, P, P_2\ MultiMergePattern(S) \wedge hasControlBlocks(S,B) \wedge hasOutputPlace(S,P_1) \rightarrow hasOutputPlace(B,P)$

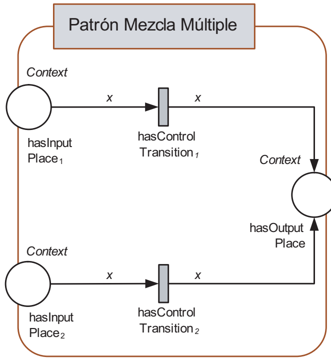


Figura 5.16: Representación del patrón mezcla

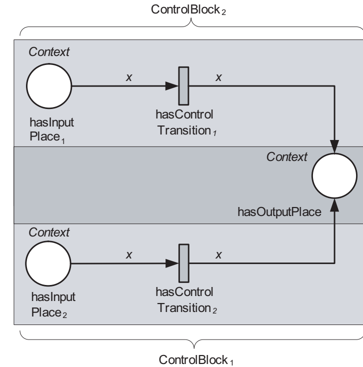


Figura 5.17: Bloques de control del patrón mezcla

La definición anterior asocia dos relaciones al concepto `MultiMergePattern`. Por una parte, la relación `hasOutputPlace` se utiliza para identificar el punto de mezcla de las ramas. Por otro lado, la relación `hasControlBlocks` apunta al conjunto de bloques de control que representan cada una de las ramas, y que se muestran gráficamente en la Figura 5.17 para una *mezcla múltiple* de dos ramas. La Tabla 5.12 completa la definición del concepto `MultiMergePattern` y añade la condición de que todos los bloques de salida compartan la plaza de salida.

La combinación de ambos patrones se muestra en la Figura 5.18 para una red de dos ramas. En este caso las plazas de salida de la *elección múltiple* se unen con las entradas de la *mezcla múltiple*. El concepto `MultiChoiceMultiMergePattern` hace referencia a este patrón:

```
subClassOf(MultiChoiceMultiMergePattern, ControlPattern).

domain(hasMultiChoice, MultiChoiceMultiMergePattern).
range(hasMultiChoice, MultiChoicePattern).
cardinality(hasMultiChoice, 1).

domain(hasMultiMerge, MultiChoiceMultiMergePattern).
range(hasMultiMerge, MultiMergePattern).
cardinality(hasMultiMerge, 1).
```

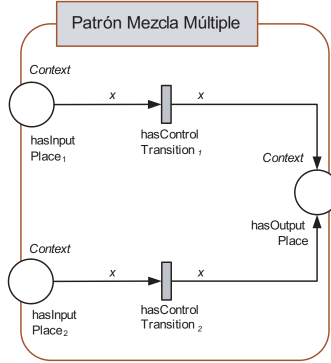


Figura 5.18: Representación de la *elección múltiple-mezcla múltiple* de dos ramas

Tabla 5.13: Axiomas que restringen el patrón *elección múltiple-mezcla múltiple*

La plaza de entrada de la red de *elección* es la misma que la plaza de entrada del patrón.
$\forall S, C, P\ MultiChoiceMultiMergePattern(S) \wedge hasMultiChoice(S,C) \wedge hasInputPlace(C,P) \rightarrow hasInputPlace(S,P)$
La plaza de salida de la red de *mezcla* es la misma que la plaza de salida del patrón.
$\forall S, M, P\ MultiChoiceMultiMergePattern(S) \wedge hasMultiMerge(S,M) \wedge hasOutputPlace(M,P) \rightarrow hasOutputPlace(S,P)$
Existe el mismo número de ramas de control en la *elección múltiple* y en la *mezcla múltiple*.
$\forall S, C, M, N_1, N_2\ MultiChoiceMultiMergePattern(S) \wedge hasMultiChoice(S,C) \wedge hasMultiMerge(S,M) \wedge cardinality(C, hasControlBlocks, N_1) \wedge cardinality(M, hasControlBlocks, N_2) \rightarrow N_1 = N_2$
La plaza de salida cada bloque de control de la *elección múltiple* será plaza de entrada de uno de los bloques de la *mezcla múltiple*.
$\forall S, C, M, N_1, N_2\ MultiChoiceMultiMergePattern(S) \wedge hasMultiChoice(S,C) \wedge hasControlBlocks(C, B_1) \wedge hasOutputPlaces(C,P) \rightarrow \exists M\ hasMultiMerge(S,M) \wedge hasInputPlaces(M,P)$

La Tabla 5.13 contiene los axiomas que complementan la semántica de esta red. Por una parte, comprueban que la plaza de entrada del patrón y de la *elección* es la misma y que la plaza de salida del patrón y de la mezcla también es la misma. Por otra parte, también se comprueba que cada uno de los bloques de control de la elección encajará con un bloque de control de la mezcla.

5.4.0.5. Patrón repetir-mientras (iterativo)

Este patrón de control describe un bucle *repetir-mientras*: comprueba la condición del bucle, *si* se cumple sigue con la ejecución del cuerpo del bucle, *sino* sale. La Figura 5.19 representa la HLPN de este patrón, en la que si la condición de la transición `hasIfTransition` se verifica, se activa la ejecución del cuerpo del bucle y se ejecuta la

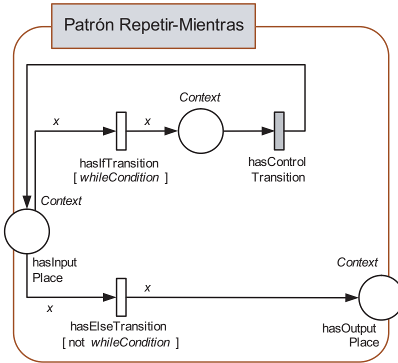


Figura 5.19: Representación del patrón repetir-mientras

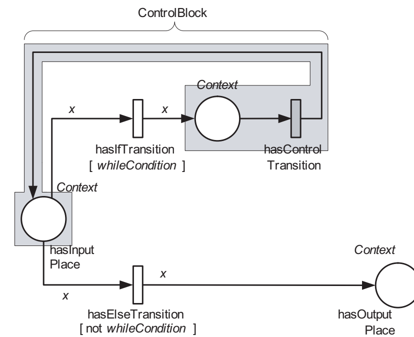


Figura 5.20: Bloques de control del patrón repetir-mientras

transición `hasControlTransition`. El arco de salida de esta transición está conectado con la plaza `hasInputPlace`, completando el proceso de iteración. Por contra, si la condición no se verifica, entonces se activa la rama alternativa y se finaliza el bucle. El concepto `RepeatWhilePattern` se refiere a este patrón:

```
subClassOf(RepeatWhilePattern, ControlPattern).

domain(hasIfTransition, RepeatWhilePattern).
domain(hasElseTransition, RepeatWhilePattern).
domain(hasControlBlock, RepeatWhilePattern).
```

El patrón *repetir-mientras* tiene una única entrada y una única salida. La plaza de entrada se identifica por medio de la relación `hasInputPlace`. Esta plaza actúa como conflicto entre las ramas *entonces* y *sino*, dado que se conecta con las transiciones `hasIfTransition` y `hasElseTransition` del patrón. Si una marca de la plaza de entrada verifica la condición de `hasIfTransition`, esta transición se activará y su disparo generará una marca en la plaza de entrada del bloque de control identificado a través de la relación `hasControlBlock`.

La ejecución de la transición de control genera una nueva marca en la plaza de entrada del patrón de forma que una nueva iteración del bucle pueda volver a iniciarse. En el caso de que la marca de la plaza de entrada no verifique la condición del bucle, la transición `hasElseTransition` se activará, y su disparo generará una marca en la plaza de salida de este patrón identificada a través de la relación `hasOutputPlace`. La Figura 5.20 identifica el bloque de control dentro de este patrón. Como se puede apreciar, la plaza de entrada del bloque es la plaza de entrada del patrón. Éste y otros axiomas de la Tabla 5.14 completan las definiciones de este patrón.

Tabla 5.14: Axiomas que restringen la semántica del patrón *repetir-mientras*

Existe un único arco entre la plaza de entrada y la transición `hasIfTransition`, y este arco está anotado con una variable.
$\forall R, P, T\ RepeatWhilePattern(R) \wedge hasInputPlace(R, P) \wedge hasIfTransition(R, T) \rightarrow uniqueArc(P, T) \wedge variableAnnotated(P, T)$
Existe un único arco entre la plaza de entrada y la transición `hasElseTransition`, y este arco está anotado con una variable.
$\forall R, P, T\ RepeatWhilePattern(R) \wedge hasInputPlace(R, P) \wedge hasElseTransition(R, T) \rightarrow uniqueArc(P, T) \wedge variableAnnotated(P, T)$
Existe un único arco entre la transición `hasIfTransition` y la plaza de entrada de la rama de control, y este arco está anotado con una variable.
$\forall R, P, T\ RepeatWhilePattern(R) \wedge hasControlBlock(R, B) \wedge hasInputPlace(B, P) \wedge hasIfTransition(R, T) \rightarrow uniqueArc(T, P) \wedge variableAnnotated(T, P)$
La plaza de salida de la rama de control es la plaza de entrada del patrón.
$\forall R, P, B\ RepeatWhilePattern(R) \wedge hasInputPlace(R, P) \wedge hasControlBlock(R, B) \rightarrow hasOutputPlace(B, P)$
Existe un único arco entre la transición `hasElseTransition` y la plaza de salida del patrón, y este arco está anotado con una variable.
$\forall R, P, T\ AnyOrderPattern(R) \wedge hasOutputPlace(R, P) \wedge hasElseTransition(R, T) \rightarrow uniqueArc(T, P) \wedge variableAnnotated(T, P)$
La transición `hasElseTransition` tiene como condición de guardia a la negación de la condición de guardia de la transición `hasIfTransition`.
$\forall R, H, S\ RepeatWhilePattern(P) \wedge hasIfTransition(P, T_1) \wedge hasElseTransition(P, T_2) \wedge hasGuard(T_1, G_1) \rightarrow hasGuard(T_2, G_2) \wedge hasOperator(G_2, boolean_not) \wedge hasArguments(G_2, L) \wedge member(G_1, L)$

5.4.0.6. Patrón repetir-hasta (iterativo)

Este patrón de control describe un bucle *repetir-hasta*: se ejecuta el cuerpo del bucle *hasta* que se cumpla una condición de salida. La Figura 5.21 representa la HLPN que modela este tipo de iteración. La principal diferencia con respecto al patrón *repetir-mientras* es que esta construcción asegura la ejecución del bucle al menos una vez. Por ello, el cuerpo del bucle se ejecuta y las condiciones se verifican al final del mismo. El concepto `RepeatUntilPattern` describe este patrón:

```
subClassOf(RepeatUntilPattern, HLPN).

domain(hasIfTransition, RepeatUntilPattern).
domain(hasElseTransition, RepeatUntilPattern).
domain(hasControlConstruct, RepeatUntilPattern).
```

La relación `hasInputPlace` se refiere a la plaza de entrada de este patrón, la cual está directamente conectada a la rama de control identificada a través de la relación `hasControlBlock` (representada en la Figura 5.22). El resultado de la ejecución del bloque de control se almacena a su vez en la plaza `hasOrSplitPlace`, que actúa como

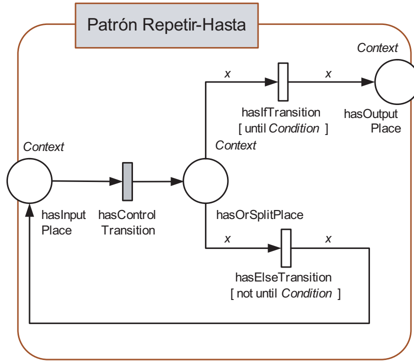


Figura 5.21: Representación del patrón repetir-hasta

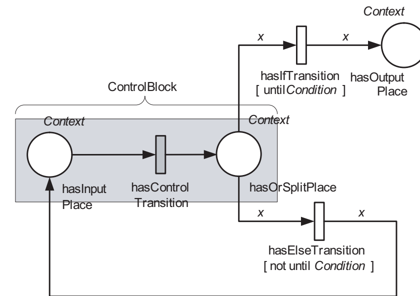


Figura 5.22: Bloques de control del patrón repetir-hasta

conflicto entre las ramas *entonces* y *sino*. La evaluación de estas ramas decidirá si se inicia una nueva iteración del bucle o no.

Las transiciones `hasIfTransition` y `hasElseTransition` manejarán en cada ciclo el desenlace del bucle. Por un lado, la relación `hasIfTransition` apunta a la transición que controla la rama *entonces* del bucle. Cuando la condición de esta transición se cumple, se ejecutará y se creará una marca en la plaza de salida del patrón. Por otro lado, la transición `hasElseTransition` está anotada con la condición del bucle, pero negada. Por lo tanto, estará activa siempre y cuando no se cumpla dicha condición, y generará una nueva marca en la plaza de entrada tras cada ejecución. Los axiomas de la Tabla 5.15 completan la definición de este patrón.

5.5. Mecanismos de composición de la red jerárquica

La composición de las distintas HLPNs que forman parte de la red jerárquica completa la visión del modelo de control. En el modelo propuesto, las páginas de la red se combinan mediante la sustitución de determinadas transiciones por otras HLPNs. Por ejemplo, la Figura 5.23 muestra gráficamente un ejemplo de sustitución en el que la transición coloreada en gris y denominada *scheduling* de la red HLPN_1 se sustituye por la red HLPN_2, dando como resultado una red jerárquica (aunque en este caso se represente como una HLPN aplanada equivalente a ella) en la parte inferior de la figura. Dado que el modelo de control restringe el tipo de redes que puede contener, es coherente pensar que el tipo de sustituciones que se pueden aplicar también esté limitado. Los tipos de sustituciones del modelo son las siguientes:

- Clase `ExecuteSubstitution`. Permite unir el patrón *proceso* y los distintos patrones de WFs que pueden formar parte de su estructura interna. Para ello, la transición *hasRunProcessTransition* del patrón *proceso* se sustituye por alguno

Tabla 5.15: Axiomas que restringen la semántica del patrón *repetir-hasta*

La rama de control comparte la plaza de entrada con el patrón.
$\forall R, P, B\ RepeatUntilPattern(R) \wedge hasInputPlace(R, P) \wedge hasControlBlock(R, B) \rightarrow hasInputPlace(B, P)$
Existe un único arco entre la plaza de salida de la rama de control y la transición `hasIfTransition`„ y este arco está anotado con una variable.
$\forall R, P, T\ RepeatUntilPattern(R) \wedge hasControlBlock(R, B) \wedge hasOutputPlace(B, P) \wedge hasIfTransition(R, T) \rightarrow uniqueArc(P, T) \wedge variableAnnotated(P, T)$
Existe un único arco entre la plaza de salida de la rama de control y la transición `hasElseTransition`, y este arco está anotado con una variable.
$\forall R, P, T\ RepeatUntilPattern(R) \wedge hasControlBlock(R, B) \wedge hasOutputPlace(B, P) \wedge hasElseTransition(R, T) \rightarrow uniqueArc(P, T) \wedge variableAnnotated(P, T)$
Existe un único arco entre la transición `hasIfTransition` y la plaza de salida del patrón, y este arco está anotado con una variable.
$\forall R, P, T\ RepeatUntilPattern(R) \wedge hasOutputPlace(R, P) \wedge hasIfTransition(R, T) \rightarrow uniqueArc(T, P) \wedge variableAnnotated(T, P)$
Existe un único arco entre la transición `hasElseTransition` y la plaza de entrada del patrón, y este arco está anotado con una variable.
$\forall R, P, T\ RepeatUntilPattern(R) \wedge hasInputPlace(R, P) \wedge hasElseTransition(R, T) \rightarrow uniqueArc(T, P) \wedge variableAnnotated(T, P)$
La transición `hasElseTransition` tiene como condición de guardia a la negación de la condición de guardia de la transición `hasIfTransition`.
$\forall R, H, S\ RepeatUntilPattern(P) \wedge hasIfTransition(P, T_1) \wedge hasElseTransition(P, T_2) \wedge hasGuard(T_1, G_1) \rightarrow hasGuard(T_2, G_2) \wedge hasOperator(G_2, boolean_not) \wedge hasArguments(G_2, L) \wedge member(G_1, L)$

de los patrones de WFs. El concepto `ExecuteSubstitution` ha sido creado para dar soporte a este tipo de sustitución:

```
subClassOf(ExecuteSubstitution, TransitionSubstitution).
allValuesFrom(hasRefinment, ExecuteSubstitution, ControlPattern).
```

Este concepto restringe a la relación `hasRefinement` para que únicamente pueda tomar un valor del tipo `ControlPattern`, es decir, un patrón del tipo *secuencia*, *separación-sincronización*, *elección exclusiva-mezcla simple*, etc. Los axiomas listados en la Tabla 5.16 se encargan de comprobar que la transición es sustituida por un patrón de WF, y que las dos fusiones que unen a los dos patrones incluyen a los nodos interfaces del patrón y a los nodos de entrada y salida de la transición sustituida.

- Clase `ControlSubstitution`. Los patrones de WFs permiten sustituir sus transiciones de control por HLPNs que se correspondan con otro patrón de WF. A través de esta composición se posibilita la creación de estructuras de control más complejas. Por ejemplo, la Figura 5.24 muestra una secuencia de dos

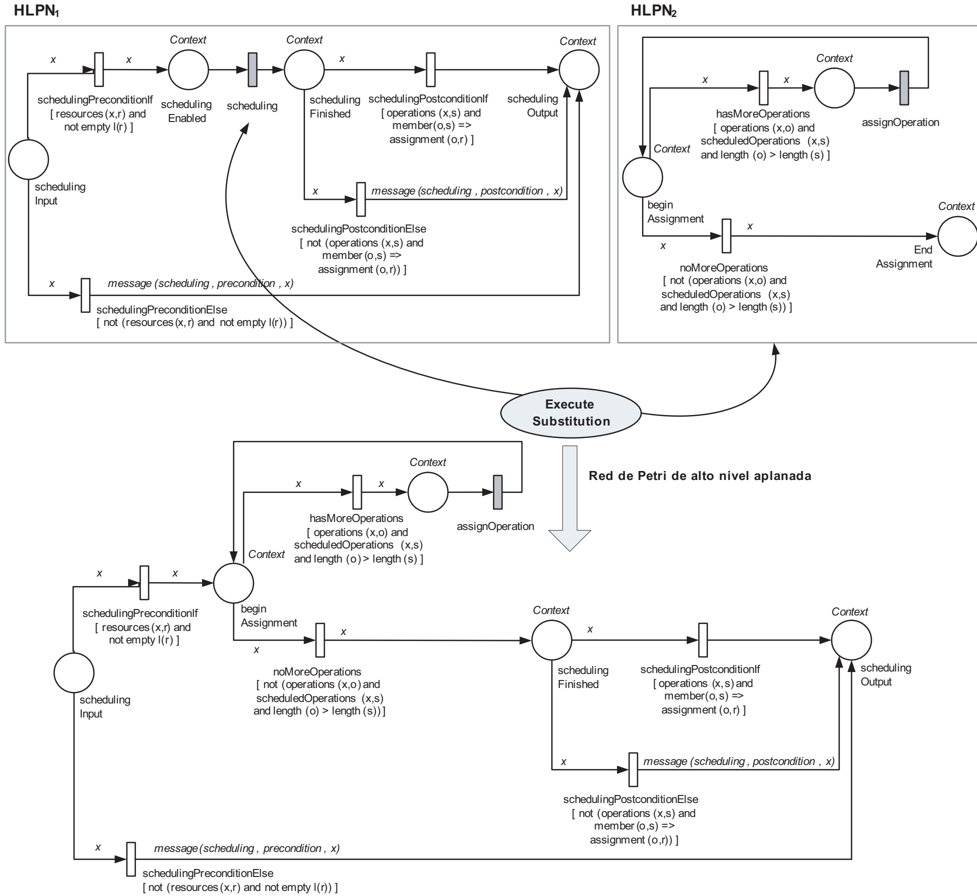


Figura 5.23: Sustitución de la transición *scheduling* por un patrón *repetir-mientras*

construcciones de control donde la primera representa un patrón *separación-unión* y la segunda una estructura iterativa *repetir-mientras*. Además no existe ninguna limitación para este tipo de sustituciones, permitiendo la definición de descripciones operacionales muy complejas. La unión entre las redes se efectúa a través de la sustitución de una transición $hasControlTransition_i$ por uno de los patrones que representan una construcción de control. El concepto `ControlSubstitution` ha sido definido para dar soporte a este tipo de sustitución:

```
subClassOf(ControlSubstitution, TransitionSubstitution).
allValuesFrom(hasRefinement, ControlSubstitution, ControlPattern).
```

Este concepto restringe la relación `hasRefinement` para que únicamente pueda tomar un valor del tipo `ControlPattern`, es decir, un patrón del tipo *secuencia*, *separación-sincronización*, *elección exclusiva-mezcla*, *elección múltiple-mezcla*

Tabla 5.16: Axiomas que restringen la semántica del concepto `ExecuteSubstitution`

El nodo sustituido es la transición *run process* del patrón *proceso*.
$\forall S, T\ ExecuteSubstitution(S) \wedge hasAbstraction(S,T) \rightarrow \exists P\ ProcessPattern(P) \wedge hasRunProcessTransition(P,T)$
Una de las fusiones debe agrupar a la plaza de entrada del patrón sustituto con la plaza de entrada de la transición sustituida.
$\forall R, S, F\ ExecuteSubstitution(S) \wedge hasRefinement(S,R) \wedge hasInterfaces(S,F) \rightarrow \exists P, P_1, P_2\ ProcessPattern(P) \wedge hasFusionSet(F,P_1) \wedge hasFusionSet(F,P_2) \wedge hasInputPlace(R,P_1) \wedge hasInputPlace(P,P_2)$
Una de las fusiones definida debe agrupar a la plaza de salida del patrón *resultado* con la plaza de salida de la transición sustituida.
$\forall R, S, F\ ExecuteSubstitution(S) \wedge hasRefinement(S,R) \wedge hasInterfaces(S,F) \rightarrow \exists P, P_1, P_2\ ProcessPattern(P) \wedge hasFusionSet(F,P_1) \wedge hasFusionSet(F,P_2) \wedge hasOutputPlace(R,P_1) \wedge hasOutputPlace(P,P_2)$

Tabla 5.17: Axiomas que restringen la semántica del concepto `ControlSubstitution`

El nodo sustituido es una transición del tipo *control construct$_i$* de un patrón de control.
$\forall S, T\ ControlSubstitution(S) \wedge hasAbstraction(S,T) \rightarrow \exists P, B\ ControlPattern(P) \wedge hasControlBlock(P,B) \wedge hasControlTransitions(B,T)$
Una de las fusiones debe agrupar a la plaza de entrada del patrón de control con la plaza de entrada de la transición sustituida.
$\forall R, S, F\ ControlSubstitution(S) \wedge hasRefinement(S,R) \wedge hasInterfaces(S,F) \rightarrow \exists P, P_1, P_2\ ControlPattern(P) \wedge hasFusionSet(F,P_1) \wedge hasFusionSet(F,P_2) \wedge hasInputPlace(R,P_1) \wedge hasInputPlace(P,P_2)$
Una de las fusiones debe agrupar a la plaza de salida del patrón de control con la plaza de salida de la transición sustituida.
$\forall R, S, F\ ControlSubstitution(S) \wedge hasRefinement(S,R) \wedge hasInterfaces(S,F) \rightarrow \exists P, P_1, P_2\ ControlPattern(P) \wedge hasFusionSet(F,P_1) \wedge hasFusionSet(F,P_2) \wedge hasOutputPlace(R,P_1) \wedge hasOutputPlace(P,P_2)$

múltiple, etc. La Tabla 5.17 contiene los axiomas que completan la definición de este concepto.

5.6. Anotación algebraica

La anotación algebraica de las HLPNs permite precisar las condiciones de activación de los eventos de la red y los efectos del disparo de dichos eventos. Por ello, la potencia de los WFs dependerá de la expresividad del álgebra que interprete las anotaciones. En este apartado se describirá la firma de estas redes, es decir, los colores que anotarán las plazas y los operadores para construir los términos, ya que el álgebra que la interpreta se asociará a través de los modelos del dominio del metamodelo.

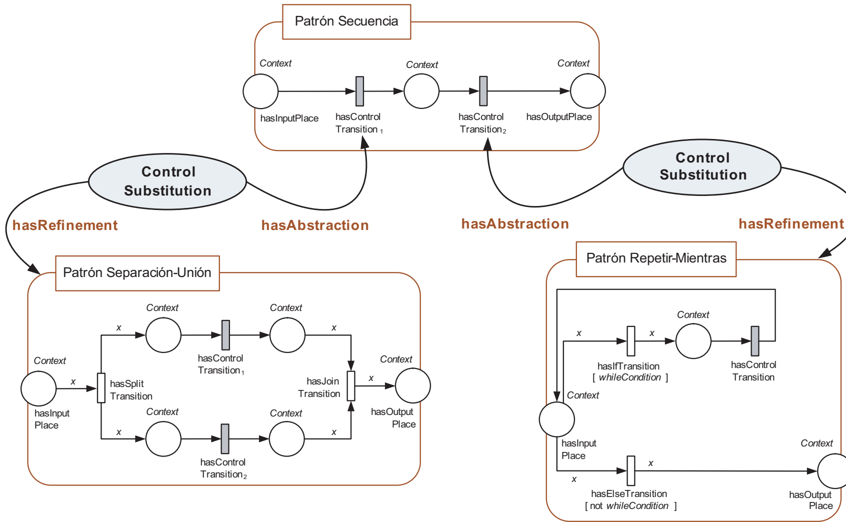


Figura 5.24: Ejemplo de sustitución de construcciones de control

5.6.1. Contexto de ejecución

En el metamodelo de WFs, el flujo de datos y de control no siempre tienen por qué coincidir. En ocasiones los PSMs no circunscriben los roles/parámetros al método sino que son accesibles desde otros método sin que exista un paso de parámetro de por medio. Esto se debe a la distinción entre los roles de conocimiento dinámicos y estáticos: los roles dinámicos, como son las entradas y salidas, sí están circunscritos al método, sin embargo, no existe un criterio para los roles estáticos. Por ejemplo, un rol estático podría utilizarse en distintos métodos y suelen representar elementos globales, como un conjunto de restricciones de fabricación y, por lo tanto, no suelen cambiar a lo largo del tiempo. Por ello cuando intervienen roles estáticos suele existir una disparidad entre el flujo de datos y el flujo de control establecido por el proceso. En estas situaciones caben dos posibles actuaciones:

1. *Definir dos HLPNs, una para los datos y otra para el control, y unirlas*. Con esta solución el diseño de la red de datos tendría una plaza por cada entrada y parámetro local y global del proceso, y otra plaza por cada salida. El valor del parámetro sería la marca situada en la correspondiente plaza. Esta solución tiene varios inconvenientes:

    - Un parámetro puede tener múltiples valores, ya que una plaza puede almacenar varias marcas.
    - No se puede usar la misma red para la ejecución de múltiples instancias del mismo proceso, ya que no se puede distinguir entre las marcas de una instancia y las de otra.

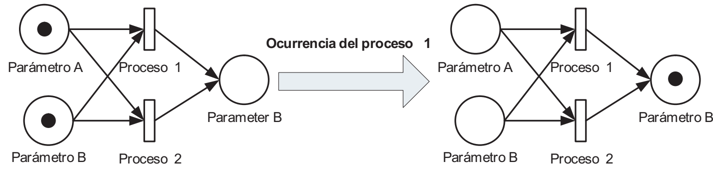


Figura 5.25: Ejemplo de red para el control del flujo de datos

- El principio de localidad de las redes de Petri.

Los dos primeros problemas se producirían únicamente si la red es reutilizada para varias ejecuciones. Estos inconvenientes pueden resolverse incrementando la complejidad de la red de forma que se pueda saber a qué instancia pertenece cada marca. El tercer problema es el más interesante, ya que está directamente ligado a la forma en la que se ejecutan las redes de Petri. Así, el disparo de una transición elimina las marcas de sus plazas de entrada y si éstas representan parámetros, entonces estos parámetros no tendrán valor después del disparo. Sin embargo, se supone que los parámetros mantienen sus valores si el proceso no los modifica explícitamente. Es más, si dos procesos comparten el mismo parámetro (la misma plaza) y uno de ellos se ejecuta, el otro nunca podrá ejecutarse. Por ejemplo, la Figura 5.25 muestra una red donde los procesos 1 y 2 comparten el mismo parámetro de entrada y el estado de la red después del disparo de la transición $proceso_1$. Se puede ver como la ejecución elimina las marcas de entrada de esa transición y genera nuevas marcas en las plazas de salida de la transición.

Después del disparo de la transición $proceso_1$, la transición $proceso_2$ deja de estar activa y, por lo tanto, ya no puede ejecutarse. Para resolver este problema, los parámetros compartidos deben *replicarse*. Por ejemplo, la Figura 5.26 representa la adaptación de la red anterior: se utilizan dos transiciones para replicar los parámetros A y B. Con esta solución el disparo de la transición $procesos_1$ no afectará a la activación y, por lo tanto, a la ejecución de la transición $proceso_2$, ya que ahora ambas transiciones no comparten ninguna plaza de entrada.

Esta solución, no obstante, también tiene un problema cuando el disparo de una transición modifica el parámetro de entrada de otro proceso. En este escenario es posible que una transición lea un valor anticuado del parámetro, es decir, realice una *lectura sucia*. Por ejemplo, la Figura 5.27 muestra una red donde el disparo de la transición $proceso_1$ producirá un nuevo valor en la plaza $parámetro_A$, pero la transición $proceso_2$ seguirá activa con el valor antiguo del parámetro A. Una solución a este problema consiste en la sincronización de los datos compartidos a través de semáforos, aunque claro está, el diseño de las redes se complica mucho.

El flujo de control de la HLPN de esta primera solución también representa cada proceso por medio de una transición, pero no incluye las plazas que representan los parámetros del proceso. En esta red las plazas representan puntos de control

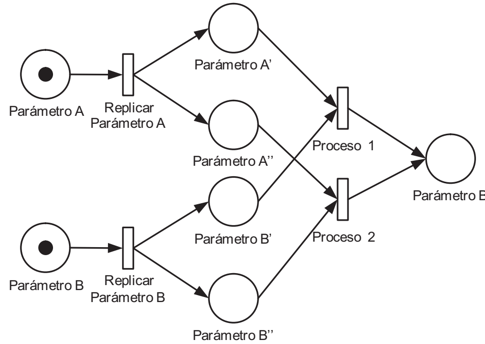


Figura 5.26: Ejemplo de red para el control del flujo de datos teniendo en cuenta el principio de localidad

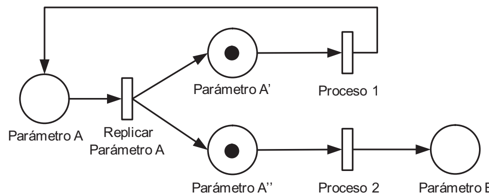


Figura 5.27: Ejemplo del problema de *lectura sucia* en una red que controla el flujo de datos

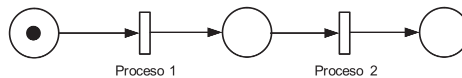


Figura 5.28: Ejemplo de red que representa un flujo de control

de forma que se pueda ordenar la ejecución de los procesos. Por ejemplo, la Figura 5.28 representa una secuencia de procesos.

La red de control y de datos de esta solución se combinan mediante la fusión de nodos: las transiciones que representan un mismo proceso en ambas redes se fusionan. Por ejemplo, la Figura 5.29 muestra un ejemplo de esta combinación. Sin embargo, la combinación de estas redes no cumple con las propiedades estructurales y de comportamiento que un WF basado en redes de Petri debería tener: no tiene una única plaza de entrada ni una única plaza de salida, no es fuertemente conexa, puede tener ciclos mortales, etc.

2. *Independizar los datos del control.* Esta solución, que a la postre fue la aplicada en esta tesis doctoral, añade complejidad a la anotación de la HLPN. Concretamente, consiste en definir un contexto de ejecución que actúa como un *bus de datos* y que contendrá cada uno de los parámetros de entrada, globales, locales y de salida que se utilizan en el proceso. Esta solución implica la definición *(i)*

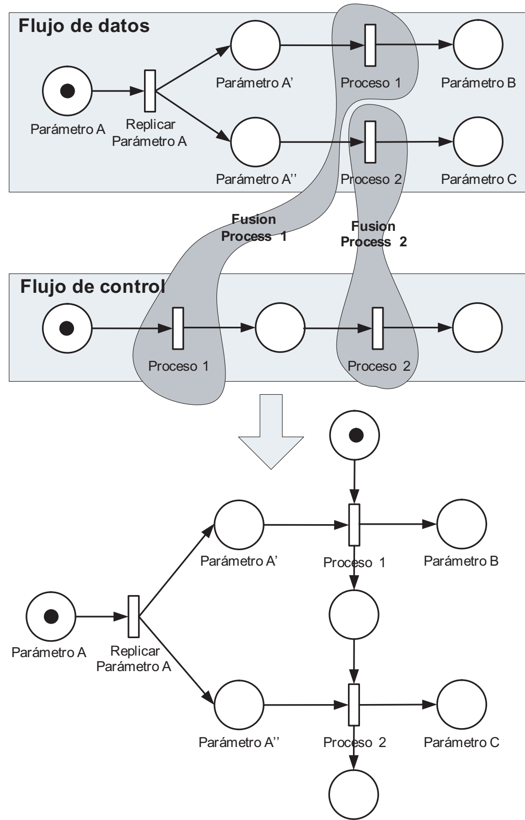


Figura 5.29: Ejemplo de combinación de una red de datos y otra de control

de un color que modele al contexto de ejecución y *(ii)* de un conjunto de operadores que representen las propiedades del contexto. Por una parte, este color anotará *todas las plazas* de las HLPNs, es decir, cada plaza de la red únicamente podrá almacenar instancias del tipo contexto de ejecución. Por otra parte, la firma contendrá un conjunto de operadores para relacionar este contexto con cada uno de los parámetros de los procesos. Por ejemplo, si el proceso que se quiere modelar tiene al $\textit{parámetro}_1$ como entrada y al $\textit{parámetro}_2$ como salida, la firma deberá contener dos operadores para acceder a los valores de sendos parámetros. Suponiendo que el tipo de dato del parámetro de entrada y de salida es *integer* y *boolean* respectivamente, estos dos operadores podrían crearse de la siguiente forma:

- $\textit{parameter}_1$: `Context` $\times$ `integer` $\longrightarrow$ `boolean`
- $\textit{parameter}_2$: `Context` $\times$ `boolean` $\longrightarrow$ `boolean`

La Figura 5.30 representa una HLPN anotada con esta segunda solución, en la que todas las plazas están tipadas con el mismo contexto de ejecución y donde

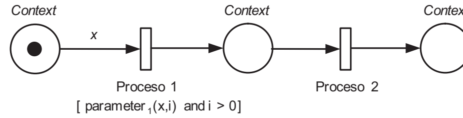


Figura 5.30: Ejemplo de la segunda aproximación para el modelado de WFs

la precondición de la transición $procesos_1$ está anotada con una expresión que contiene el operador anterior. La evaluación de este operador permitirá acceder, a través del contexto de ejecución almacenado en la variable x, a un valor para la variable i que verifique la condición.

Al trabajar siempre sobre el mismo contexto, las transiciones pueden acceder a cualesquiera datos y tomar sus decisiones en función de ellos. Sin embargo, aunque el contexto de ejecución sea el mismo, los valores de los parámetros con los que está relacionado se pueden modificar tras la ejecución de una transición. Por ejemplo, el disparo de una transición puede tomar el contexto de ejecución de su plaza de entrada y modificar un parámetro de salida de dicho contexto.

Con esta solución se evitan los problemas de la propuesta anterior: no es necesario replicar nodos, definir semáforos, ni fusionar transiciones en las estructuras de control. Comentar que los niveles de composición del proceso no afectan a esta solución, ya que los identificadores de cada uno de los parámetros dentro del WF son únicos. Esto significa que dentro de un WF no puede haber dos variables distintas con el mismo identificador. Además, permite utilizar la red para la ejecución de varias instancias del proceso, ya que los identificadores de los contextos serán distintos para cada ejecución. Finalmente, es importante resaltar que, al contrario de la aproximación anterior, las redes resultantes cumplen con las propiedades que un WF basado en redes de Petri debería tener.

5.6.2. Firma sintáctica

La firma sintáctica contiene las bases para la anotación de las HLPNs: define los colores que anotarán las plazas y los operadores que anotarán los arcos. La firma que anota las HLPNs debe cumplir con algunos requisitos. Uno de esos requisitos, relacionado con el apartado 5.6.1, es incluir el color `Context` que representa al contexto de ejecución que anotará las plazas de las HLPNs. A nivel sintáctico, el contexto de ejecución es una simple declaración:

```
hasSorts(ControlSignature, Context).
Sort(Context).

hasName(Context, "Context").
hasName(Context, "Represents an abstract execution context").
```

Cabe mencionar que este color anotará todas las plazas de las HLPNs indicando de esta forma que todas las marcas que se vayan a localizar en las plazas de estas redes

serán *contextos de ejecución*:

$$\forall R, H, P\ ControlModel(C) \wedge Page(H) \wedge Place(P) \wedge \\ \wedge\ hasPages(C, H) \wedge hasNodes(H, P) \rightarrow hasSort(P, Context)$$

También es necesario comentar que no tiene sentido refinar este color y adecuarlo a un determinado proceso, ya que la firma únicamente describe elementos sintácticos, por lo que un subcolor de `Context` no añadiría nada. Además, el paso para adecuarlo a un determinado problema se realiza a través del tipo de dato que interpreta al contexto dentro del álgebra.

Aunque el contexto de ejecución no es el único color definido dentro de la firma sintáctica sí es el único obligatorio:

$$\forall R, H, S\ ControlModel(C) \wedge Page(H) \wedge Signature(S) \wedge \\ \wedge\ hasPages(C, H) \wedge hasSignature(H, S) \rightarrow hasSorts(S, Context)$$

Además, cualquier firma incorporará por defecto algunos colores que representan los tipos básicos usados en programación. Por ejemplo, se incluyen los números enteros, números flotantes, símbolos, cadenas de texto, tipo lógico, además de los tipos que permiten tratar las fechas y el tiempo:

```
hasSorts(ControlSignature, integer_sort).
hasSorts(ControlSignature, float_sort).
hasSorts(ControlSignature, symbol_sort).
hasSorts(ControlSignature, string_sort).
hasSorts(ControlSignature, boolean_sort).
hasSorts(ControlSignature, date_sort).
hasSorts(ControlSignature, time_sort).
hasSorts(ControlSignature, dateTime_sort).
hasSorts(ControlSignature, duration_sort).
```

Los demás colores son añadidos en función de las necesidades del proceso. Estos colores, al igual que los que representan los tipos básicos, se usarán para especificar a los rangos y dominios de los operadores de la firma. En este sentido, también se incluyen a los operadores más significativos de cada uno de los tipos básicos. Por ejemplo la firma contendrá al operador de suma, resta, multiplicación, división y demás operadores de comparación de números enteros y flotantes, operadores de conjunción, disjunción, negación e igualdad, así como algunos operadores para tratar con el tiempo y las fechas:

```
hasOperators(ControlSignature, integer_addition).
hasOperators(ControlSignature, integer_subtraction).
...
hasOperators(ControlSignature, true).
hasOperators(ControlSignature, false).
hasOperators(ControlSignature, boolean_and).
...
```

Estos operadores son representaciones sintácticas y, por lo tanto, no tienen asociada una implementación. Por ejemplo, el operador que realiza la suma de números enteros se definiría de la siguiente forma:

```
Operator(integer_addition).
hasName(integer_addition, "integer_addition").
hasDomain(integer_addition, [integer_sort, integer_sort]).
hasRange(integer_addition, integer_sort).
```

Dadas las características de este metamodelo, surge la cuestión de cómo trabajar con las instancias de una ontología dentro de las HLPNs. A lo largo de esta tesis, cada uno de los elementos de una ontología ha sido definido a través de un predicado: las instancias de una clase mediante un predicado unario mientras las instancias de una relación a través de un predicado binario. Esta misma solución es también aplicable a las HLPNs simplemente incluyendo los predicados como operadores con el rango de tipo lógico. Por ejemplo, para comprobar que un elemento es del tipo `Context` basta con definir el siguiente operador:

```
Operator(Context).
hasName(Context, "Context").
hasDomain(Context, [symbol_sort]).
hasRange(Context, boolean_sort).
```

Puede apreciarse como el operador `Context` toma un símbolo como entrada y devuelve un valor de tipo lógico. En este caso, el parámetro del operador representará el símbolo que identifica la instancia. El mismo procedimiento es aplicable a la definición de un operador que represente una relación. Por ejemplo, para comprobar si el contexto de ejecución tiene el parámetro *username* bastaría con definir el siguiente operador:

```
Operator(username).
hasName(username, "username").
hasDomain(username, [Context, symbol_sort]).
hasRange(username, boolean_sort).
```

Las HLPNs harán uso de estos operadores a través de los términos que anotan las transiciones y los arcos. Por ejemplo, la expresión representada en la Figura 5.31 podría perfectamente anotar la precondición de una transición. En este caso, el término es la conjunción de varios operadores donde dos de ellos son los previamente definidos. A través de esta condición, la transición puede restringir el disparo de la transición a aquellos usuarios cuyo nombre de usuario tenga el formato de una dirección de correo electrónico.

A través de estos mecanismos, se puede operar con las instancias de una ontología. Remarcar la importancia de estos operadores, ya que el contexto de ejecución contendrá una propiedad por cada parámetro definido en el proceso. Por lo tanto, existirá un operador para acceder a cada parámetro que pueda contener el contexto de ejecución.

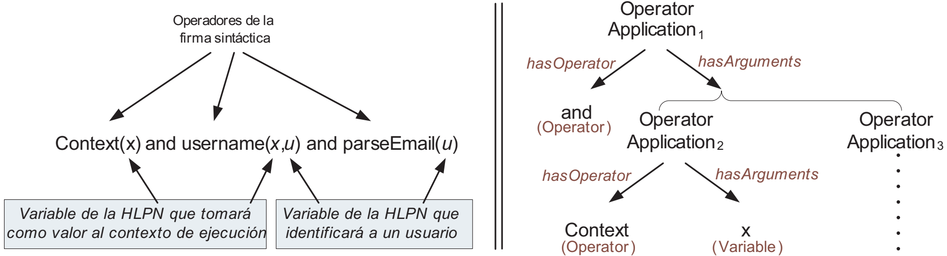


Figura 5.31: Ejemplo de un término que representa la llamada un operador

5.7. Conclusiones

En este capítulo hemos descrito en detalle la parte del metamodelo que permite coordinar la ejecución de las tareas que componen un WF: el modelo de control, que es una de las principales aportaciones de esta tesis doctoral en relación a cómo otros metamodelos enfocan descripción operacional de los métodos. En este sentido, y al contrario de otras aproximaciones, el control se modela de forma que:

- La estructura de los WFs, y por consiguiente la descripción operacional de los métodos compuestos, se representa a través de redes de Petri jerárquicas, un formalismo aceptado por la comunidad de WFs y muy utilizado para la representación de procesos de negocio en cualquier ámbito de aplicación. El uso de un formalismo de representación de procesos tiene la ventaja de permitir el diseño de modelos que no sean ambiguos, además de dotar de semántica operacional a la mayoría de los comportamientos típicos de cualquier proceso de negocio (por ejemplo, la concurrencia).

- Las redes de Petri del modelo de control están explicitadas a través de la ontología propuesta en el Capítulo 3. Como ya hemos comentado, esta ontología permite describir tanto las características estructurales como las características dinámicas de las redes de Petri, de tal forma que cuando se crea una red el diseñador puede conocer con exactitud el comportamiento de su modelo. Además, la explicitación mediante una ontología posibilita el razonamiento acerca de las características de los elementos que constituyen la firma y el álgebra de la red, que en nuestro metamodelo serán ontologías de un determinado dominio de aplicación.

- El modelo de control también aporta los conceptos que permiten describir los patrones de WFs. Por ejemplo, existe un concepto `SequencePattern` que representa al patrón de WFs *secuencia*, y que permite especificar la ejecución ordenada de un conjunto de tareas. La semántica operacional de cada uno de estos patrones está basada en un determinado modelo de red de Petri. Para conjugar la semántica de la ontología con la del modelo de redes de Petri, cada concepto extiende al concepto `HLPN`, es decir, cada concepto es una red de Petri,

y además tiene asociado un conjunto de axiomas que garantizan que, tanto la estructura de la red, como su anotación, cumplen el patrón.

En resumen, en este apartado se ha propuesto una nueva forma de diseñar la descripción operacional de los métodos compuestos basada en el formalismo matemático de las redes de Petri y en los patrones de WFs. Esta aproximación tiene ciertas ventajas respecto a otras existentes en la bibliografía científica:

- *Soporte formal al modelado de procesos.* Las pocas propuestas que incorporan la descripción operacional de un método a su metamodelo [186] suelen modelar la coordinación entre las tareas a través de lenguajes lógicos, que no están basados en una lógica concurrente. Estas propuestas describen la coreografía a través de constructores de control, al igual que se implementa un programa tradicionalmente, pero no asocian una semántica operacional a dichos constructores. En este sentido es necesario que un programa externo interprete sus constructores con el consiguiente problema de ambigüedad.

- *Traslada la idea de diseño paramétrico del metamodelo a la descripción operacional.* El modelo de control posibilita la construcción de un proceso como si de un rompecabezas se tratara. El modelo de control estará formado por un conjunto de HLPNs, que representan las piezas del rompecabezas, y por un conjunto de operadores de composición, que representan las uniones entre las piezas. Por ejemplo para diseñar un proceso con una estructura interna formada por la secuencia de un patrón *repetir-mientras*, *separación-unión* y *elección-mezcla* bastaría con crear las instancias de la *secuencia*, el *repetir-mientras*, la *separación-unión* y la *elección-mezcla* e indicar que la primera transición de la secuencia se sustituye por el *repetir-mientras*, etc.

- *Facilita la reutilización de las descripciones operacionales.* Otra de las ventajas de la ontología de redes de Petri utilizada para modelar el control es que independiza *(i)* la estructura de la red *(ii)* de su anotación y *(iii)* de su comportamiento. De esta forma y a través de los refinadores del modelo de control resulta relativamente sencillo extender o modificar una determinada parte de la red. Además, al contrario que los demás metamodelos de conocimiento, el propuesto en esta tesis doctoral independiza el control de los métodos. Así, una misma red podría utilizarse para describir el comportamiento de varios métodos.

Una vez descrito el metamodelo de representación de WFs y el modelo de control que es el núcleo de dicho metamodelo, es necesario desarrollar e implementar una infraestructura tecnológica que permita la definición y ejecución de los componentes de conocimiento que forman parte del WF. Esta infraestructura se describirá en detalle en el Capítulo 6.

6

Infraestructura para la ejecución de flujos de trabajo

Hasta este momento, se han descrito los conceptos que componen el metamodelo y que nos permiten modelar flujos de trabajo (WFs, del inglés *Workflow*). Por un lado, en el Capítulo 4 se ha visto que el metamodelo tiene una arquitectura poco acoplada, formada por componentes de conocimiento independientes que describen cada uno de los aspectos de un WF. Por el otro, en el Capítulo 5 se ha detallado cómo la descripción operacional de los WFs se representa a través de redes jerárquicas construidas combinando patrones de control hasta dar con el comportamiento deseado. En este capítulo se irá un paso más allá y se describirá en detalle la infraestructura tecnológica [270, 274] que da soporte a la construcción y ejecución de WFs. Su arquitectura está representada en la Figura 6.1, y se compone de cuatro capas que proporcionan los mecanismos para *(i)* describir cada uno de los componentes del WF, *(ii)* componer el modelo de ejecución, *(iii)* comunicar el sistema con los distintos agentes que participan en la ejecución, y *(iv)* facilitar la integración de los agentes dentro del sistema de WFs.

6.1. Descripción de los flujos de trabajo

La primera capa proporciona los medios para gestionar cada uno de los componentes del metamodelo, adaptarlos en caso de ser necesario, y relacionarlos entre sí a través de puentes. Como se ha descrito en el Capítulo 4, cada uno de estos componentes describe un aspecto particular e independiente del WF y su unión mediante puentes permite generar un modelo al nivel del conocimiento del problema a resolver.

6.2. Composición del modelo de ejecución

La segunda capa de la infraestructura da soporte a la ejecución de WFs. Esta parte constituye el núcleo del WMS y es el punto donde se decide qué actividad ejecutar y quién estará a cargo de su ejecución. Las principales funciones de esta capa son las siguientes:

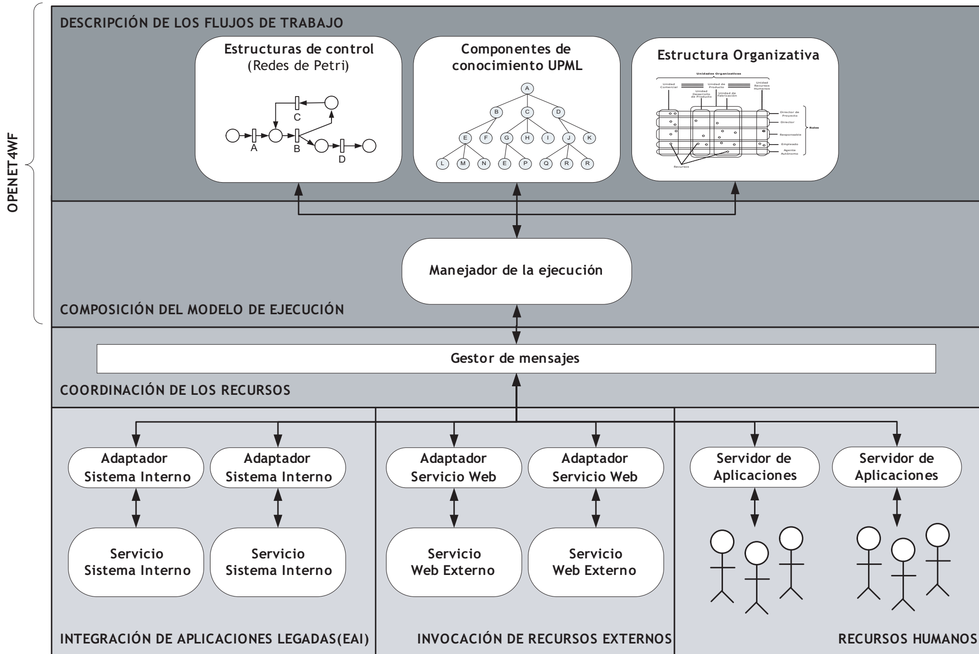


Figura 6.1: Arquitectura de la infraestructura tecnológica para la ejecución de WFs

- *Construir el modelo de ejecución.* Se deriva del modelo de WF construido por el diseñados. Es un proceso computacionalmente costoso que requiere la composición de los distintos modelos de control del WF en una red jerárquica y la transformación de dicha red en una HLPN directamente ejecutable.

- *Dar soporte a la ejecución.* Ya que el modelo de ejecución es directamente interpretable, esta capa se encarga de gestionar la ejecución de este tipo de redes. Para ello, dispone de un motor de HLPNs a través del cual se deducen las transiciones activas y los modos de ejecución para cada una de ellas. Esta capa se encarga de seleccionar el modo de transición a ejecutar y de solicitar al motor de HLPNs su ejecución. Un tipo especial de transiciones con las que debe lidiar esta capa son aquellas que están anotadas para ser ejecutada por algún recurso. Su finalización no se producirá hasta que alguno de los recursos que pueden realizarla no le comunique al sistema su ejecución y el resultado de la misma. A partir de entonces, esta capa se encargará de actualizar el estado de la HLPN a partir de la información suministrada por el recurso.

- *Gestionar las listas de trabajo.* Esta capa también se encarga de hacer visible las asignaciones de trabajo a los distintos recursos por medio de una lista de trabajos pendientes. De esta forma los recursos podrán saber en cualquier instante los trabajos que tienen asignados.

6.2.1. Construir el modelo de ejecución

Nuestro metamodelo especifica un WF a través de un conjunto de componentes de conocimiento y de puentes que los relacionan entre sí. Aunque este modelo define todos y cada uno de los aspectos de un WF, no es en sí un modelo ejecutable, sin embargo, provee las bases para construirlo. El punto de partida para llevar a cabo esta construcción son los métodos y en particular el método que resuelve el WF. Dejando de lado a los métodos primitivos a cargo de los recursos, los métodos complejos se resuelven a través de una estructura de control descrita mediante el formalismo de las redes de Petri. Aun así, la red que resuelve un método compuesto no es completa: un componente de control detalla la coordinación de las (sub)tareas que componen el método, pero, no profundiza en la resolución de cada una de estas tareas. Consecuentemente, para dar una solución completa es necesario establecer un vínculo entre todos los métodos que participan en la solución del mismo problema.

Por otro lado, tal y como se explicó en el Capítulo 5, las redes de Petri detalladas en el modelo de control carecen de álgebra, ya que sólo están descritas a través de la firma sintáctica. Esto quiere decir que las anotaciones de las transiciones y de los arcos de la red sólo son *etiquetas*. Por ejemplo, una transición anotada con la expresión $mayor(x, y)$ no tendrá efecto en el sistema hasta que no tenga asignado un comportamiento para el operador $mayor$. Desde el punto de vista del metamodelo, el álgebra de la red no está incluida en el modelo de control para mejorar su reutilización: sin álgebra, un mismo modelo de control puede ser la descripción operacional de varios métodos. Por lo tanto, uno de los pasos imprescindibles para construir el modelo de ejecución es dotar de álgebra a las HLPNs del modelo de control, y consecuentemente al modelo de ejecución. Ahora bien, la semántica que han de tomar los operadores que anotan las redes ya está recogida en el metamodelo a través de los métodos, modelos del dominio y modelos de recursos. Así, los puentes establecen las correspondencias entre el modelo de control y el método, entre el modelo de control y el del dominio, y entre el método y el modelo de recursos, en realidad están añadiendo el conocimiento necesario para asociar el álgebra a la red.

Existen más incógnitas relacionadas con la semántica operacional, por ejemplo, ¿cómo se incorpora el conocimiento al proceso? Si la ejecución está a cargo de redes de Petri, ¿cómo se asocian las correspondencias entre la dimensión de procesos y conocimiento? Lo mismo sucede con los recursos, ¿cómo se incorporan los recursos al modelo de redes de Petri? En este apartado se tratará de dar respuesta a todas estas preguntas detallando la construcción del modelo que da soporte a la ejecución de WFs.

6.2.1.1. Crear la red jerárquica de ejecución

La funcionalidad de los WFs está recogida en las *tareas* del metamodelo. Por ello, ante la necesidad de resolver un problema a través de un WF, el primer paso consiste en buscar una tarea dentro de la librería del sistema que modele o represente el mismo tipo de problema o, en su defecto, crear una nueva tarea. La funcionalidad expresada

por una tarea suele ser genérica dado que está expresada en términos independientes del dominio y de la solución. Por ejemplo, una tarea asociada al proceso de compra de un material puede enfocarse desde perspectivas diferentes en función del departamentos de la empresa en el que se realice: aunque en cada departamento se resuelva la tarea de compra, la solución adoptada por cada uno puede ser distinta.

La solución a una tarea está especificada a través de la combinación del método y de su modelo de control. Por ello, el segundo paso consiste en seleccionar uno de los métodos capacitados para resolver la tarea, es decir, aquel cuyas características funcionales se aproximen más a los de la tarea. En cualquier caso es necesario resaltar que aunque las entradas del método y las de la tarea no coincidan, ello no descarta el uso de dicho método para la resolución de la tarea: esta disparidad en las entradas podría resolverse a través del conjunto de correspondencias introducidas por el puente que vaya a unir al método con la tarea.

Si el método seleccionado es primitivo, la tarea ya tendría solución. Sin embargo, en caso de ser un método compuesto, contendría otras (sub)tareas a las cuales sería necesario buscar una solución. Por ejemplo, la Figura 6.2 representa lo que se denomina un *árbol de descomposición de tareas*. La raíz de este árbol es la tarea que describe el problema y el segundo nivel representa el método encargado de resolverla. Este método es compuesto y por ello tiene un conjunto de hijos, cada uno de los cuales representa una (sub)tarea a resolver que a su vez se soluciona mediante un método. De esta forma el árbol no estará completo hasta que todas las tareas han sido asociadas al método que las resuelve. Como resultado de este proceso tanto las tareas como los métodos compuestos son los nodos del árbol, mientras que los métodos primitivos son las hojas. La unión o asignación entre las tareas y los métodos se realiza a través del puente *tarea-método* que detalla las correspondencias necesarias para unir ambos componentes.

Una vez que todos los métodos han sido seleccionados, el tercer y último paso consiste en elegir el modelo de control más adecuado para representar la descripción operacional de cada uno de los métodos compuestos del árbol de descomposición de tareas. Como se ha descrito en el Capítulo 4, este modelo especifica una estructura de coordinación a través de una HLPN jerárquica que se detalla independientemente del método que vaya a ejecutar y de las tareas en las que dicho método se descompone. Por ello, la el modelo de control se define exclusivamente en términos de las redes de Petri y sólo entiende de páginas, plazas, transiciones, arcos, etc. En este sentido, la unión mediante un puente *método-control* no sólo asocia la terminología entre ambos componentes, sino también las (sub)tareas del método con las transiciones de las HLPNs y las precondiciones con las anotaciones de la red, etc. Por ejemplo, la Figura 6.3 muestra la integración del control dentro del diagrama de descomposición de tareas de la Figura 6.2. Puede verse en la figura como los métodos complejos se sustituyen por sus correspondientes HLPNs y como algunas transiciones representan directamente una de las tareas en las que se descompone el método.

En realidad la integración es más compleja, ya que la Figura 6.3 sólo muestra la parte que se refiere a la asociación tareas-métodos. Como se vio en el Capítulo 5, el modelo de control es una red jerárquica compuesta por un conjunto de HLPNs donde

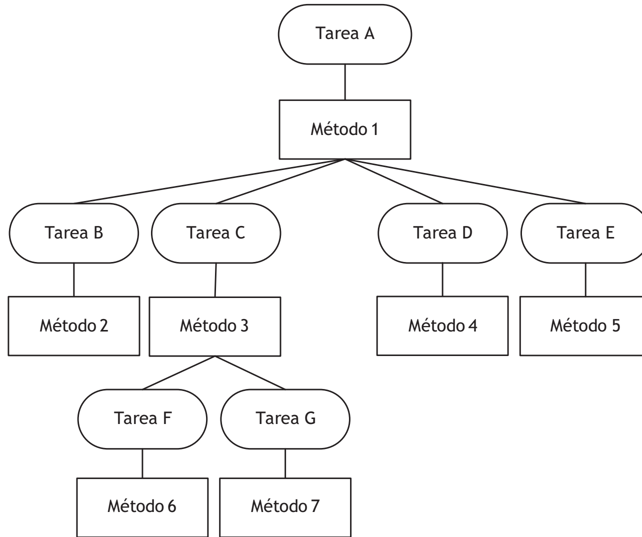


Figura 6.2: Ejemplo de diagrama de descomposición de tareas

cada HLPN representa un determinado patrón de WFs. Por ejemplo, la HLPN de la Figura 6.4 muestra todas las HLPNs que forman parte de la red jerárquica que controla el *método 1*. La primera de estas redes (HLPN0) representa al patrón proceso que, como se comentó en el Capítulo 5, es el nodo raíz del modelo de control. En este patrón se muestran varias anotaciones que aluden a la integración entre el método y el modelo de control. Por una parte, la anotación a la izquierda identifica a la condición de guardia de la transición *ifPrecondition* que contiene al operador *precondition*. El puente entre el método y el control establecerá una correspondencia en la que se asocia al operador *precondition* la función del álgebra que lo interpretará. Por otra parte, la anotación a la derecha aplica las correspondencias respecto a la postcondición del método. Finalmente, la última clase de correspondencias establecidas entre el método y el modelo de control identifica las (sub)tareas del método con las transiciones del modelo de control. Por ejemplo, en la Figura 6.4 las transiciones que representan una tarea están resaltadas en las redes HLPN1, HLPN2 y HLPN3.

De forma más detallada, los pasos a seguir están recogidos en el algoritmo de la Figura 6.5:

1. El primer paso consiste en crear la instancia de la red jerárquica h que representará el modelo de ejecución. Una vez creada, la red se irá completando tomando como punto de partida la tarea principal que describe la funcionalidad del WF, es decir, la raíz del árbol de descomposición de tareas.

2. El segundo paso consiste en identificar el método que resuelve la tarea principal del WF, y para ello se usa el puente *tarea-método* que relaciona la tarea con un método m en el modelo del WF. Si m es primitivo, la estructura de la red

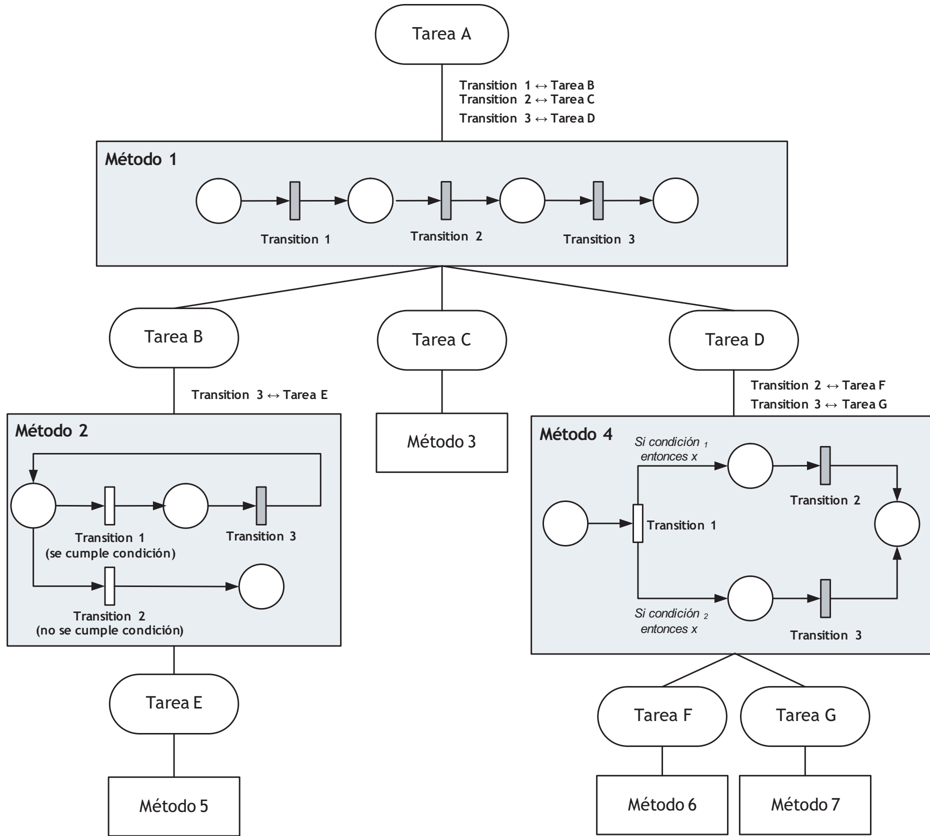


Figura 6.3: Integración del control dentro del diagrama de descomposición de tareas

jerárquica no se vería afectada, ya que estos métodos no tienen descripción operacional al estar su ejecución a cargo de un recurso. En cambio, si m es compuesto será necesario integrar su descripción operacional dentro de la red jerárquica de ejecución.

3. El paso 3.a identifica el modelo de control que coordina la ejecución de las (sub)tareas del método compuesto. Al existir una única descripción operacional asociada al método, este paso consiste en localizar el puente que relaciona el método con un modelo de control. Finalmente, el paso 3.b invoca el procedimiento *Añadir* encargado de completar la red jerárquica h, para lo cual se agrega *(i)* cada uno de los componentes del modelo de control c, y *(ii)* cada uno de los modelos de control c_i que resuelven algunas de las (sub)tareas del método m. En un primer paso este procedimiento añade las páginas y operaciones de composición (sustituciones y fusiones) de c. Después, incorpora, en caso de ser necesario, a cada uno de los modelos de control asociados a cada una de las (sub)tareas del método. Esta acción se corresponde con el paso 3 del procedimiento *Añadir*, y

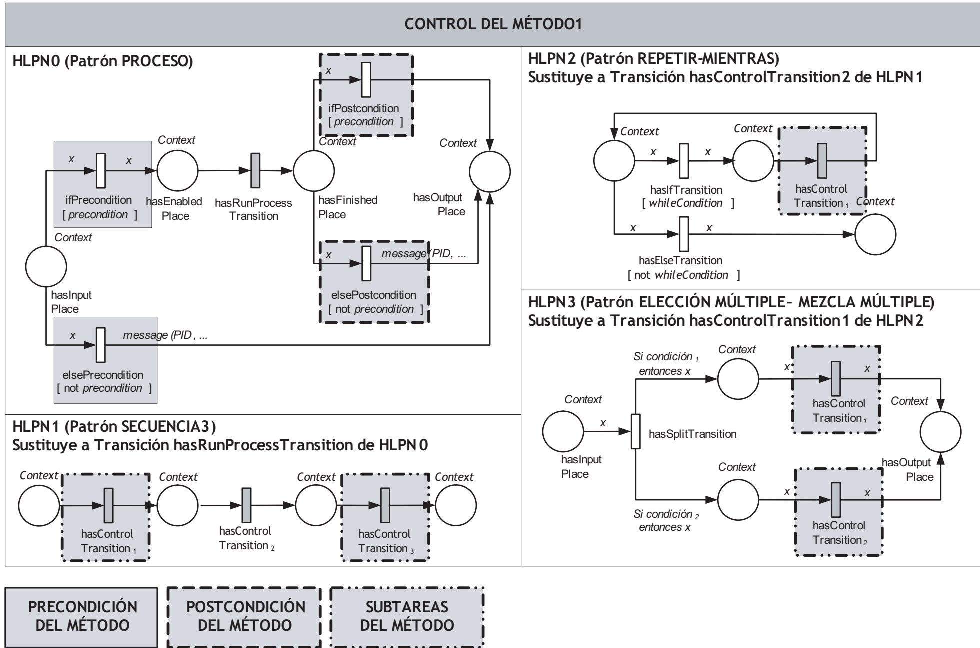


Figura 6.4: HLPNs del modelo de control del método 1

se fundamenta en un bucle que procesa cada una de las correspondencias entre una transición de c y una (sub)tarea del método m, y que se encarga de buscar los métodos m_i que resuelven las (sub)tareas y de agregar su control a la red jerárquica h. Si el método m_i es compuesto, su modelo de control c_i se añade a h a través de una nueva invocación del procedimiento *Añadir*. Finalmente, los pasos 3.b.3 y 3.b.4 permiten conectar los distintos modelos de control entre sí a través de una sustitución entre la transición que representa la tarea en c y el patrón proceso p_i del modelo de control c_i que la resuelve.

Puente método-control. Además de asociar un método a un modelo de control es necesario *(i)* relacionar la descomposición en tareas del método con determinadas transiciones de la red de control, y *(ii)* asociar las precondiciones y postcondiciones del método en la estructura de la red. Por ejemplo, para el caso de las precondiciones se trata de crear una función que permita interpretar al operador *preconditions* que anota la condición de guardia de la transición `ifPreconditionTransition` del patrón de proceso. Como se puede apreciar en la Figura 6.6, la correspondencia permite establecer que la expresión de esa función se obtiene de las precondiciones del método. Partiendo de la representación gráfica es difícil intuir que las precondiciones del método puedan definir el cuerpo/expresión de una función. Sin embargo, esto es viable porque, como se puede apreciar en la figura, las precondiciones se representan

Principal

1. Crear HLPN jerárquica h.
2. Identificar el método m que resuelve la tarea principal t.
3. Si el método m es compuesto:
    a) Identificar el modelo de control c que resuelve m.
    b) *Añadir* (c, m, h).

Añadir (c, m, h)

Si m es compuesto y no ha sido añadido previamente a h:

1. Añadir las páginas, sustituciones y fusiones de c a la red h.
2. Identificar las correspondencias que relacionen una transición de c con una (sub)tarea de m:
3. Para cada correspondencia $cor(transition, task)$:
    a) Identificar el método m_i que resuelve la tarea $task$.
    b) Si el método m_i es compuesto:
        1) Identificar el modelo de control c_i que resuelve el método m_i.
        2) *Añadir* (c_i, m_i, h).
        3) Identificar la página p_i que representa el patrón proceso de c_i.
        4) Crear nueva sustitución entre la página p_i y la transición $transition$.

Figura 6.5: Creación de la estructura jerárquica del modelo de ejecución

a través de una fórmula/regla aplicable. En lo referente a las postcondiciones se aplicaría el mismo tipo de razonamiento, pero en este caso se interpretaría el operador $postcondition$.

6.2.1.2. Integración de los recursos

En una empresa suele ser usual asociar un modelo de recursos para cada WF de la organización. Aunque cada uno de estos modelos puedan llegar a compartir el mismo modelo de organización (definido a través de una instancia de la ontología TOVE), las reglas de asignación no tienen por qué coincidir. Además, no todos los WFs son creados para las mismas personas y una misma persona puede jugar papeles diferentes y tener responsabilidades distintas en varios WFs. Por ejemplo, un WF para la compra de material suele incorporar únicamente al personal del departamento de compras. En cambio, un WF para el presupuestado de un producto suele involucrar a personal de todas las áreas de la empresa, al tratarse de una tarea transversal que toca a la mayor parte de sus divisiones.

La intersección entre la dimensión de recursos y la de procesos asocia los agentes a los procesos, de modo que los métodos primitivos son el punto de unión entre ambas dimensiones. El puente *método-recursos*, que conecta ambos componentes entre sí, permitirá seleccionar aquellos recursos con derecho a ejecutar el método. Esta

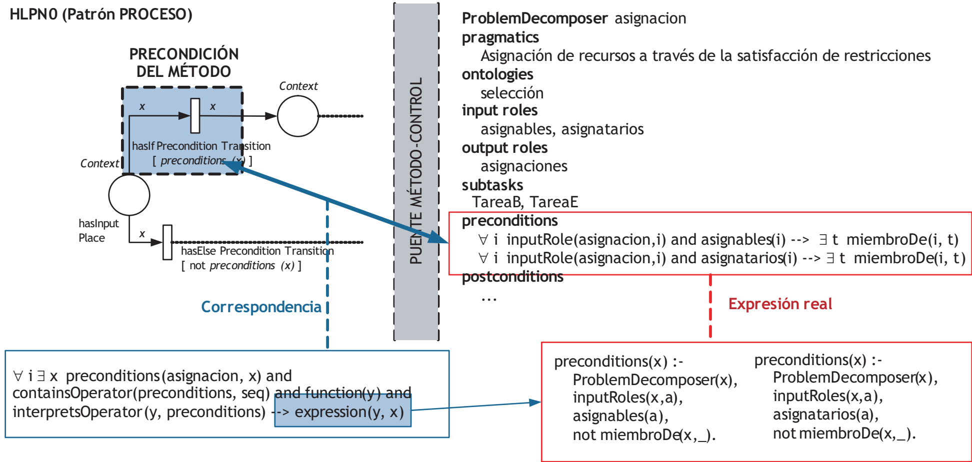


Figura 6.6: Precondiciones de un método primitivo en el álgebra del modelo de ejecución

selección podrá ser directa o estar basada en alguna de las agrupaciones previamente definidas en el modelo de la organización, como por ejemplo los equipos, roles, permisos, divisiones, etc. Estos criterios de asignación han de incorporarse dentro de las condiciones de guardia de la transición asociada a la tarea que debe resolver el método. La Figura 6.7 muestra un ejemplo de intersección entre las dimensiones de procesos y de recursos que asocia el método primitivo *método2* con la *división1* y el *rol3*. Esta correspondencia permite inferir que cualquier agente x que cumpla la condición $performs(x, actividad2)$ podrá ejecutar el método, como por ejemplo el $agenteA$.

En definitiva, ya que los métodos primitivos carecen de implementación (estarían fuera del nivel de conocimiento), se considera que los recursos están al cargo de los detalles de su ejecución. De esta forma, se desliga la implementación de los métodos del metamodelo y se delega su ejecución a los recursos. Esta separación también permite analizar el rendimiento de los recursos: por ejemplo, si dos recursos están habilitados para ejecutar el mismo método primitivo, el WMS podría implementar una función que analizase el quehacer de cada recurso e incorporar dicho conocimiento como un criterio más de selección.

Puente método-recursos. La asociación entre métodos y recursos también afecta al álgebra del modelo de ejecución. Como se señaló a lo largo de este capítulo, los métodos primitivos no tienen asociada ninguna descripción operacional y, por lo tanto, se representan mediante una transición en el modelo de ejecución. La ejecución de esta transición estará a cargo de un determinado recurso que será asignado a partir de un criterio de selección establecido en el puente entre métodos y recursos. Este puente permite incorporar estos criterios de selección al álgebra del modelo de ejecución y a la

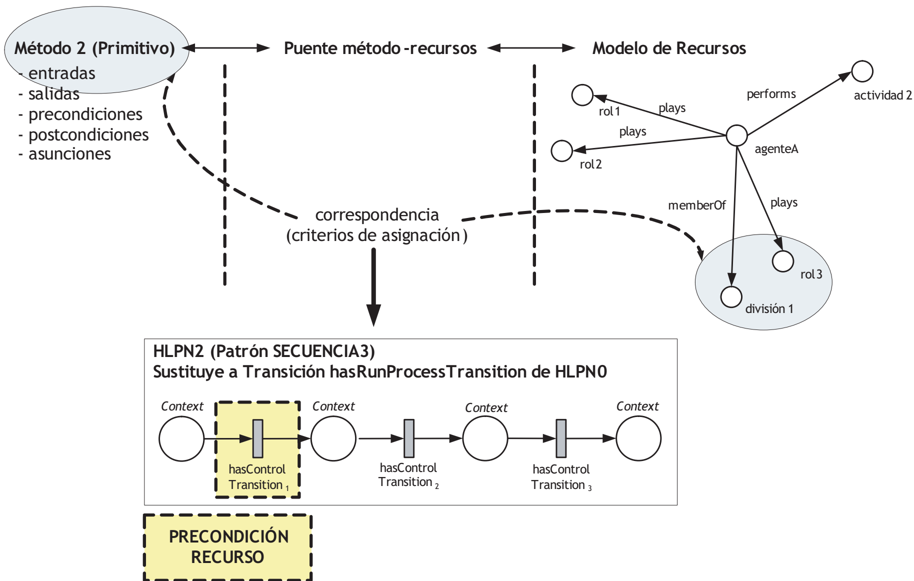


Figura 6.7: Las transiciones que se resuelven mediante un método primitivo añadirán la precondición del recurso

propia red. Por una parte, el álgebra incluirá una función cuya expresión contendrá las reglas previamente introducidas en la propiedad `knowledge` del modelo de recursos. La labor de esta función será interpretar la condición de guardia de la transición que representa al método primitivo dentro del modelo ejecución. Por ejemplo, la Figura 6.8 muestra gráficamente las distintas correspondencias que nos permiten establecer la precondición de la transición que ejecutará el método primitivo.

Asignación de recursos. El metamodelo propuesto especifica los criterios de asignación de trabajo en el modelo de recursos. Por lo tanto, cuando un diseñador construye un WF, está aplicando estos criterios a los métodos primitivos a ejecutar. Ya que estos criterios se integran dentro del modelo, la propia dinámica de la HLPN indicará durante la ejecución qué recursos pueden realizar las actividades.

Como se ha visto en la Figura 6.8, los criterios de asignación se integran como condiciones de guardia de las transiciones que representan a métodos primitivos. Así, la activación y ejecución de estas transiciones estará restringida a aquellos recursos que cumplan con los criterios de asignación. Por ejemplo, supóngase el siguiente modelo de recursos expresado en TOVE:

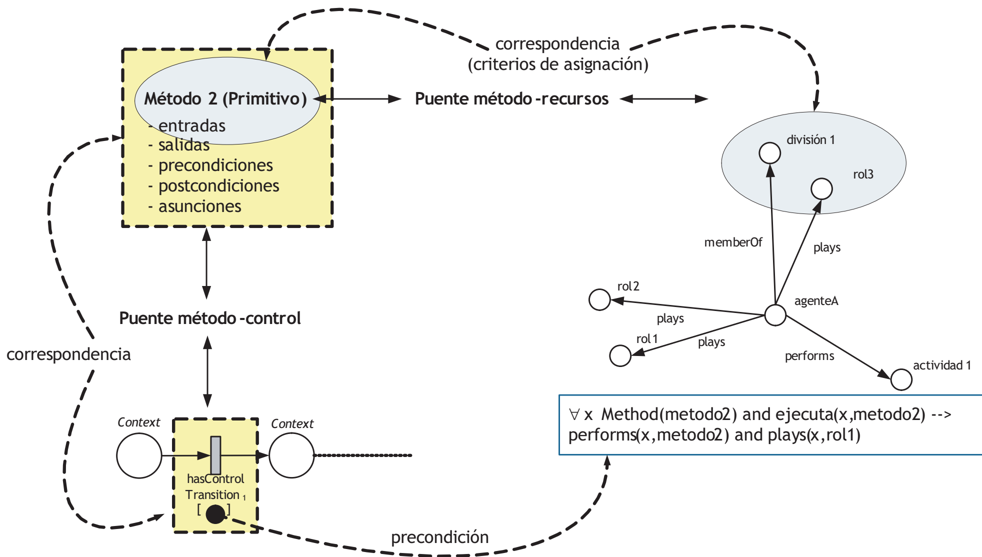


Figura 6.8: Correspondencias entre los componentes de conocimiento y el álgebra de la red

```
Role(director)              Role(profesor)              Role(investigador)
Agente(alberto)             Agente(juan)                Agente(manuel)
Agente(pedro)
Actividad(ac1)              Actividad(ac2)              Actividad(ac3)
performs(alberto, ac1)      performs(alberto, ac2)      performs(alberto, ac3)
performs(manuel, ac2)       performs(manuel, ac3)       performs(juan, ac2)
performs(pedro, ac2)
plays(alberto, director)    plays(alberto, profesor)    plays(alberto, investigador)
plays(manuel, director)     plays(manuel, profesor)     plays(manuel, investigador)
plays(juan, director)       plays(pedro, director)
...
```

Partiendo de la red representada en la Figura 6.8, supóngase también que la plaza de entrada de la transición t_1 contiene una marca $contexto_1$ con el contexto de ejecución. En esta situación, el motor evaluaría la condición de guardia $Context(x) \wedge Agent(a) \wedge Role(director) \wedge hasAgent(x,a) \wedge plays(a, investigador)$ de la transición para cada uno de los valores que puedan tomar las variables x y a. Aquellos valores para los cuales se verifique la expresión constituyen un modo de transición, y además se dirá que la transición está activa. Para la base de conocimiento anterior los modos de transición serían $(contexto_1, alberto)$, $(contexto_1, manuel)$, $(contexto_1, juan)$, $(contexto_1, pedro)$, indicando que los usuarios *alberto*, *manuel*, *juan* y *pedro* están habilitados para ejecutar el método. Por lo tanto, esta entrada se añadirá a la lista de trabajos pendientes para cada uno de los agentes y se actualizará continuamente a partir de los modos de transición de los métodos primitivos que estén activos. Es necesario puntualizar que cuando uno de estos métodos se ejecuta *(i)* uno de estos modos de transición es seleccionado, *(ii)* se elimina la marca de la plaza de entrada

del método y *(iii)* se genera una nueva marca con el nuevo contexto de ejecución en la plaza de salida. Al no tener marcas en su entrada, la transición asociada al método dejará de estar activa y, por consiguiente, desaparecerá de las listas de trabajos pendientes de los recursos.

6.2.1.3. Integración del conocimiento del dominio

Existen distintos criterios a la hora de seleccionar o crear el modelo del dominio de un WF. Algunas empresas prefieren usar un único modelo del dominio para así poder compartirlo y reutilizarlo en todos los WFs. El problema de esta estrategia es que da lugar a dominios muy genéricos, que además suelen contener mucho más conocimiento del que es necesario para ejecutar el WF que se está diseñando. La otra estrategia consiste en crear un dominio que contengan únicamente el conocimiento imprescindible para ejecutar el WF de interés. El problema de esta solución es que *trocea* el conocimiento de un dominio. No obstante, ambas soluciones son correctas: la primera mantiene el dominio más coherente, aunque tiene un menor rendimiento, ya que tiene que inferir conocimiento innecesario. En cambio, la segunda es menos exigente desde un punto de vista computacional, dado que infiere únicamente el conocimiento imprescindible para solucionar el WF.

Una vez seleccionado el modelo del dominio, el siguiente paso consiste en unirlo con las dimensiones de proceso y de recursos. Así, por una parte, la intersección entre las dimensiones de conocimiento y de procesos se realiza a través de dos puentes: *tarea-dominio* y *método-dominio*. El primer puente contiene la mayor parte de las correspondencias entre la terminología del dominio y la de los procesos. Como ya se comentó en el Capítulo 4, cuando un dominio establece relaciones con una tarea, también se está relacionando transitivamente con los métodos a través del puente *método-tarea*. Por ello, el puente *método-dominio* sólo se encarga de asegurar que el dominio cumple con las asunciones establecidas en el método.

Puente dominio-control. El puente entre el modelo del dominio y el modelo de control permite establecer las correspondencias entre la terminología usada para anotar los términos de las HLPNs y el conocimiento de la aplicación. De los capítulos 3 y 5, se sabe que la terminología utilizada para anotar las HLPNs está formada por un conjunto de operadores y tipos de datos agrupados en una misma firma sintáctica y que dicha terminología se adquiere de las ontologías utilizadas por el modelo de control. Por ejemplo, si la ontología utilizada para anotar la red de control trata la fabricación de muebles, entonces la firma sintáctica incorporará al menos tipos de datos como: $Mueble$, $Máquina$, $Recurso$, etc. Asimismo, incorporará operadores que relacionan dichos tipos, como por ejemplo $tienePiezas : Mueble \rightarrow Pieza$, $utilizaRecursos : Máquina \rightarrow Recurso$, etc.

Por lo tanto, los tipos de la firma sintáctica serán identificadores de clase de las ontologías mientras que los operadores serán predicados que permiten inferir las clases y relaciones de dichas ontologías. Además, la firma también podría incorporar nuevos tipos y operadores no incluidos en las ontologías, como por ejemplo aquellos relacio-

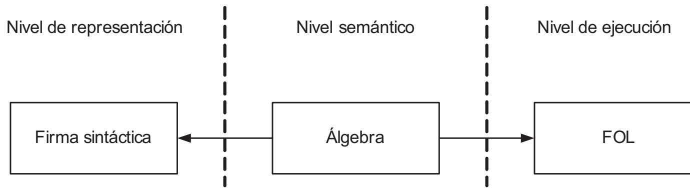


Figura 6.9: Los tres niveles de la anotación algebraica

nados con el contexto de ejecución (o canal de ejecución) y utilizados para anotar las plazas de las HLPNs del modelo de control. Al existir tan estrecha relación entre los operadores que anotan las HLPNs y las ontologías, las correspondencias entre ambos modelos es prácticamente directa. Por ejemplo, un predicado del dominio de la aplicación denominado *mayorQue* podría asumir la interpretación del operador *mayor* previamente definido en la firma. El modelo de control contendrá el comportamiento asociado al predicado *mayorQue*, cuya semántica estará declarada dentro de las fórmulas asociadas a la propiedad `knowledge`. En el caso del predicado *mayorQue*, el cálculo estará a cargo de las siguientes reglas:

```
mayorQue(?_x, ?_y, true) :-
  ?_x > ?_y.
mayorQue(?_x, ?_y, false) :-
  ?_x <= ?_y.
```

La solución descrita anteriormente muestra la potencia de tener un álgebra asociada a una firma sintáctica. Por un lado el álgebra permite reutilizar redes con dominios de aplicación diferentes. Por otro lado, permite reutilizar redes con motores de inferencia diferentes que tengan lenguajes de representación diferentes simplemente adaptando las reglas del dominio al lenguaje correcto. Finalmente mencionar una última ventaja: con esta aproximación no es necesario crear un intérprete de los términos que anotan las redes de Petri, ya que la evaluación recaerá en el motor de inferencia que se encarga de ejecutar las redes. De esta forma, el propio razonamiento está desligado de las HLPNs haciendo estas redes más reutilizables. La Figura 6.9 representa esta separación donde el álgebra actúa como intermediario entre la sintaxis de anotación de las redes y su semántica de interpretación.

6.3. Gestor de mensajes

La tercera capa de la infraestructura facilita las comunicaciones y la interacción entre los recursos que participan en la ejecución del WF. Dependiendo del tipo de recurso, humano o software, la interacción con el manejador de la ejecución puede variar. Lo mismo sucede con quién ha de tomar la iniciativa de la comunicación. Por ejemplo, si el recurso es un servicio web, el manejador debería tomar la iniciativa e invocar la ejecución de ese servicio. En cambio, si el recurso es humano, el usuario estará a

cargo de la comunicación, es decir, accederá al WMS, recuperará su lista de tareas pendientes y decidirá cuál de ellas ejecuta. En este último caso, la comunicación de los resultados también será responsabilidad del usuario.

Con el fin de independizar la infraestructura del tipo de comunicación que pueda establecerse entre el sistema y un determinado recurso, la tercera capa se basa en el paradigma de *paso de mensajes*, que es la solución ideal para dar soporte a comunicaciones distribuidas y poco acopladas. Es el tipo de comunicación que mejor se adapta a la dinámica de ejecución de los WFs, ya que al contrario que las llamadas a procedimientos remotos (RPC, del inglés *Remote Procedure Call*), no necesitan que la comunicación entre el cliente y el servidor esté sincronizada. En este sentido, en los RPCs cuando una parte se comunica con la otra, se ha de esperar hasta que el método invocado finalice su ejecución; es decir, se requiere que tanto el cliente como el servidor estén disponibles al mismo tiempo. La comunicación es, por lo tanto, poco adecuada cuando se quiere desacoplar las aplicaciones. En cambio, el pase de mensajes relaja este tipo de comunicación introduciendo un módulo intermedio, llamado cola, que permite que los distintos componentes software se relacionen entre sí de forma asíncrona. El principal beneficio de este tipo de comunicación es que evita que el remitente necesite tener un conocimiento preciso de los receptores, ya que la interacción se realiza a través de la cola.

El paso de mensajes utilizado por esta infraestructura se basa en un modelo de publicación/suscripción cuyos tópicos identifican los recursos. Así, los suscriptores se registrarán en el sistema con el fin de recibir los mensajes destinados a un determinado recurso que son enviados por la instancia del WF. Como consecuencia, en este modelo ni el publicador ni el suscriptor se conocen de forma que no existe acoplamiento alguno entre ambas entidades. Además, el modelo también se caracteriza por:

- Permitir múltiples suscriptores de un mismo recurso. Con esta configuración es posible que varios suscriptores compitan por las tareas de un mismo recurso o que se repartan el trabajo. En cualquier caso, estas estrategias de trabajo colaborativo no están contempladas en la infraestructura.

- Los mensajes publicados mientras el suscriptor no está conectado son redistribuidos una vez que se conecta. Así, no se pierde ningún mensaje y la lista de trabajos pendientes está siempre actualizada.

La Figura 6.10 muestra de forma gráfica la arquitectura del modelo de mensajería, donde se puede ver cómo las instancias de los WFs envían los mensajes referentes a un determinado tópico y los suscriptores reciben dichos mensajes. La principal característica del modelo es que los tópicos se refieren a los identificadores de los recursos y los suscriptores son, por lo tanto, las implementaciones de los propios recursos, que se encargarán de ejecutar los métodos primitivos del WF.

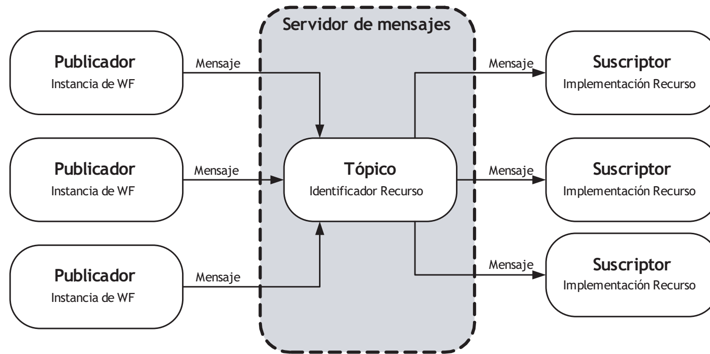


Figura 6.10: Modelo de publicación/suscripción del repartidor (broker) de mensajes

6.4. Participantes del flujo de trabajo

La cuarta capa integra en la infraestructura de ejecución de WFs a *(i)* sistemas de legado existentes en la organización, *(ii)* servicios externos, y *(iii)* otros agentes software. Esta integración se lleva a cabo mediante adaptadores que actúan como suscriptores del gestor de paso de mensajes. A través de esta solución cada adaptador registrado en esta capa representará a uno de los recursos que participa en la ejecución del WF. De esta forma, los recursos se tratan uniforme e independientemente de su implementación.

Cada adaptador debe implementar las funcionalidades básicas que le permitan interpretar los mensajes del gestor. Por ejemplo, cuando el manejador de la ejecución le comunica al recurso que tiene una nueva actividad asignada, el adaptador deberá recuperar la definición del método del WMS; es decir, sus entradas, salidas, y pre/postcondiciones de la misma. De la misma forma, el adaptador debe notificar los resultados al manejador de acuerdo con la firma del método; es decir, deberá asociar un valor a cada uno sus parámetros (entradas y salidas).

Aunque desde la perspectiva del WMS los adaptadores son indistinguibles, hacia fuera del sistema cada adaptador deberá manejar el protocolo de comunicaciones, el formato de los mensajes, el lenguaje de descripción, etc. del recurso en cuestión. Por ejemplo, cuando un adaptador integra a un servicio web, debe saber cómo invocarlo. Para ello, primero ha de tener el fichero WSDL (del inglés, *Web Service Description Language*) que describe dicho servicio o en su defecto saber cómo recuperar la referencia a ese fichero de un registro de servicios UDDI (del inglés, *Universal Description, Discovery and Integration*). Segundo, ha de saber cómo definir las correspondencias entre la firma del método primitivo y la del servicio web externo. Para entradas y salidas especificadas a través de algún tipo básico, es decir, para los enteros, números reales, valores lógicos, etc., el paso es prácticamente directo. Sin embargo, para tipos complejos requerirá una lógica más compleja. Finalmente, debe saber cómo invocar el servicio web a través de SOAP (del inglés *Simple Object Access Protocol*) al proveedor de servicios. La Figura 6.11 muestra este escenario, en el que se puede ver

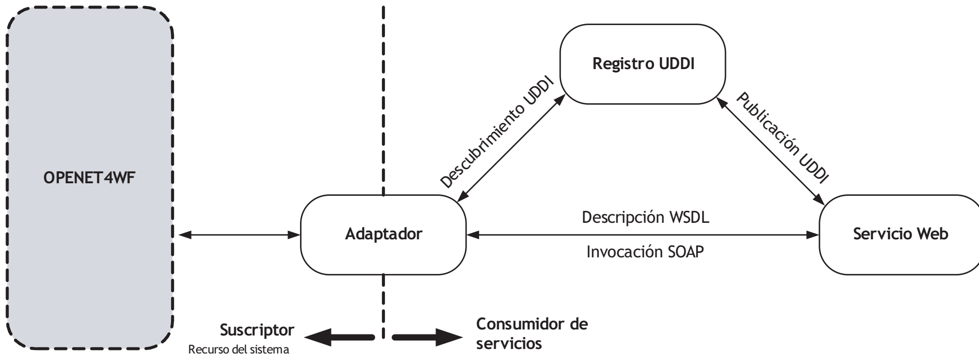


Figura 6.11: Adaptador para la integración de servicios web

como el adaptador es el punto de unión entre el WMS y los servicios web externos y cómo interacciona con ambos sistemas. Por un lado es el suscriptor del WMS y por el otro el consumidor de servicios web. En lo tocante a su faceta de consumidor de servicios, se ve cómo se relaciona con el registro UDDI de cara a descubrir el servicio. Internamente, el adaptador dispone de una tabla que le permite relacionar los métodos primitivos con los servicios web a invocar (un adaptador/recurso puede dar soporte a varios métodos primitivos). La figura también muestra cómo a partir de la descripción WSDL del servicio, el adaptador es capaz de comunicarse con el servicio web a través del intercambio de mensajes en SOAP.

Los demás tipos de adaptadores se construyen aplicando el mismo procedimiento. Por ejemplo, la Figura 6.12 muestra un adaptador para la integración de aplicaciones legadas a través de CORBA (del inglés *Common Object Request Broker Architecture*). En este caso, la conexión entre el WMS y el sistema legado se realiza a través de llamadas a procedimientos remotos mediante CORBA. Al igual que en el caso de los servicios web, el adaptador se encarga de la integración entre ambos sistemas, es decir, de traducir los requerimientos del WMS al lenguaje del sistema integrado (en este caso a IDL - *Interface Definition Language*) e invocar los procedimientos a través de CDR (del inglés, *Common Data Representation*) bajo el protocolo de comunicaciones IIOP TCP (del inglés, *Internet Inter-ORB Protocol TCP*).

6.5. OPENET4WF: Motor de flujos de trabajo

Una implementación software (OPENET4WF) [270, 274] da soporte al modelo descrito en el Capítulo 4. Dado que este modelo basa su operacionalidad en términos de HLPNs, sería lógico pensar que un motor estándar de HLPNs bastaría para dar soporte a su ejecución. Sin embargo, no existen motores que permitan ejecutar HLPNs descritas a través de la ontología propuesta en esta tesis doctoral. Se podría pensar en traducir las instancias de la ontología de HLPNs a otro formato, por ejemplo al formato de intercambio del estándar ISO/IEC 15909-2, pero como se ha visto en el

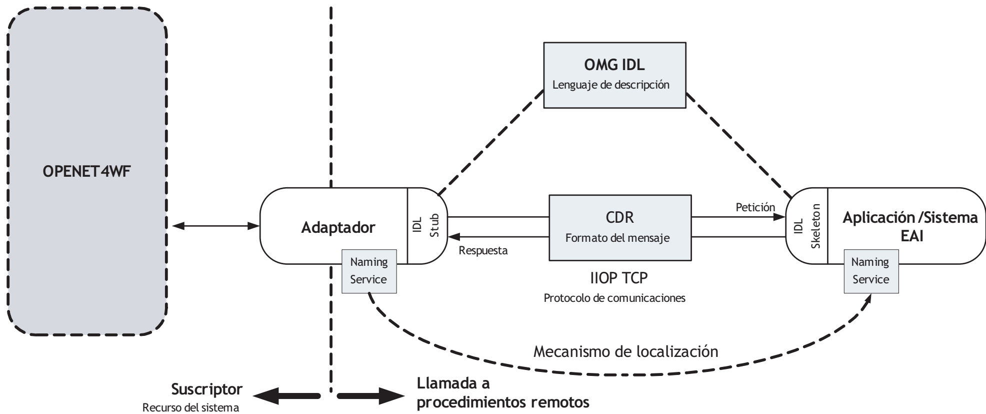


Figura 6.12: Adaptador para la integración de sistemas legados a través de CORBA

Capítulo 3, ningún formato alternativo permite introducir el álgebra asociada a la red, es decir, sólo permiten intercambiar grafos. Una segunda solución consistiría en pasar la ontología al formato propietario de alguna de las herramientas de redes de Petri disponibles en el mercado. Aunque la mayoría de estas herramientas están pensadas para la simulación, alguna de ellas posibilita traducir su representación gráfica a otro lenguaje de programación (por ejemplo C++) a partir del cual realizar la ejecución. Si bien esta solución es aplicable para la ejecución de una red de Petri, no lo es para un WF: a través de estas herramientas no es posible integrar los agentes que participan en la ejecución y, por lo tanto, se necesitaría extender a través de la programación el código asociado al WF. Además, se perderían muchas de las ventajas de nuestra propuesta: no sería posible razonar sobre las características de la ejecución ni compartirla entre distintos agentes. Una componente especialmente relevante de OPENET4WF es el motor OPENET encargado de la ejecución de HLPNs descritas mediante la ontología propuesta en el Capítulo 3 [276]. El motor está construido sobre una base de conocimiento para facilitar la introducción de nuevo conocimiento y de ontologías. Además integra un razonador lógico para razonar sobre el conocimiento del dominio y sobre las características propias de las HLPNs. El resto de OPENET4WF se ha construido partiendo de esta arquitectura y añadiendo distintas capas al motor de HLPNs para dar así soporte a las redes jerárquicas, a los patrones de WFs y finalmente a los componentes del metamodelo.

6.5.1. OPENET: Motor de HLPNs

La Figura 6.13 muestra la arquitectura del motor OPENET que da soporte a la ejecución de HLPNs [276]. El núcleo de este motor es la dualidad formada por la ontología de HLPNs y el razonador. La primera homogeneiza la definición de estas redes y además simplifica su validación, ya que contiene el conocimiento que describe semánticamente los elementos de una HLPN, evitando la implementación de un módulo

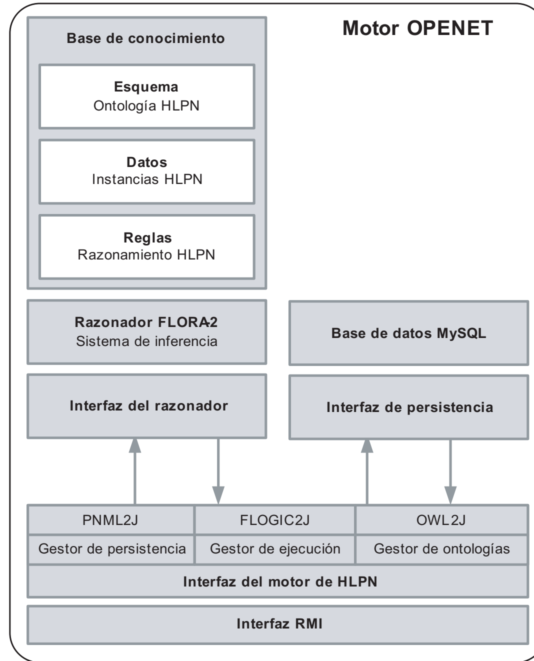


Figura 6.13: Arquitectura del motor OPENET. El motor está construido sobre el razonador FLORA-2 al que añade soporte de persistencia.

software específico. El segundo gestiona la ontología de HLPNs, realiza la validación del grafo y de la ejecución, y permite razonar acerca de las instancias de la ontología de HLPNs y de aquellas ontologías del dominio usadas para anotar la red. De forma más detallada, OPENET se compone de los siguientes elementos:

- *Base de conocimiento*. Es un tipo especial de base de datos para la gestión de conocimiento (base de datos deductiva), aunque sin soporte de persistencia entre ejecuciones. La base de conocimiento se compone de los siguientes elementos:

    - *Esquema*. Almacena el conjunto de clases, relaciones, propiedades y axiomas que describen las redes de Petri. Describe semánticamente los componentes estáticos (plazas, transiciones, arcos, términos, etc.) y el comportamiento dinámico y está basada en el estándar matemático ISO /IEC 15909-1 [144] en el que están especificadas las HLPNs.

    - *Datos*. Este módulo almacena las instancias de las redes de Petri y de los modelos del dominio usados para anotar la red. Por ejemplo, las clases de las ontologías del dominio se insertan como metaclases del álgebra utilizada para anotar las redes de Petri. Estas clases, además de usarse para la creación de instancias del dominio, también se usan para definir ope-

```
%doFiring(?_exe, ?_tml, ?_fir) :-
   ?_exe:HLPNExecution,
   ?_exe[currentMarking -> ?_cma, Firings -> ?_fil],
   %areTransitionModesConcurrent(?_exe, ?_tml),
   %areTermsEvaluatedInTransitionModes(?_tml),
   %createStep(?_tml, ?_ste),
   %copyMarking(?_cma, ?_tma),
   %createFiring(?_ste, ?_cma, ?_tma, ?_fir),
   append(?_fil, [?_fir], ?_nfl)@_prolog(basics),
   %updateHlpnExecution(?_exe.HLPN, ?_exe.initialMarking, ?_tma, ?_nfl, ?_exe),
   %fireStep(?_fir).
```

Figura 6.14: Ejemplo de ejecución de un multiconjunto de modos de transición. En este caso, el predicado `%doFiring` toma por entradas a la red en ejecución ?_exe, a la lista de modos de transición ?_tml a ejecutar y devuelve en la variable ?_fir una instancia del concepto *Firing* que denota el cambio de estado de la red.

radores; es decir, un operador podrá referirse a una de estas clases en su dominio o rango.

- *Reglas*. Esta fachada incorpora los mecanismos que permiten razonar sobre las HLPNs. Por una parte, añade a la base de conocimiento un conjunto de reglas tanto para el manejo de las redes de Petri como de las ontologías del dominio descritas en F-Logic [152], las cuales permiten crear, modificar y borrar las instancias de la ontología de redes de Petri. Por ejemplo, sirven para autogenerar los identificadores/namespaces de las instancias que se crean en la base de conocimiento. Por otro lado, esta fachada también implementa las reglas para la automatización de parte del comportamiento de las redes. Por ejemplo, la Figura 6.14 muestra el predicado que se encarga de ejecutar un multiconjunto de modos de transición. Para ello comprueba que los modos son concurrentes, que existen las evaluaciones de los arcos de entrada y salida de las transiciones a ejecutar y, si se cumplen los requisitos anteriores, de realizar el cambio de estado de la red. Además, esta fachada proporciona las reglas que permiten obtener aquellos términos cuya evaluación debe realizarse fuera del razonador, preguntar si varios modos de transición son concurrentes en el estado actual, volver a un determinado estado de la red, etc.

- *Razonador FLORA-2*. FLORA-2[1] es un sistema basado en conocimiento y un entorno completo para el desarrollo de aplicaciones que hacen uso de conocimiento. FLORA-2 extiende el cálculo de predicados clásico con el concepto de objeto, clase y tipo, los cuales han sido adaptados de la programación orientada a objetos. Este sistema se basa en F-Logic e integra la programación lógica y las bases de datos deductivas dentro del paradigma orientado a objetos. FLORA-2

[1] http://flora.sourceforge.net

```
%error(HLPN_1, ?_net, ?_arc) :-
   ?_net:HLPN,
   ?_net[arcs -> ?_arc],
   ?_arc[sourceNode -> ?_sno, targetNode -> ?_tno],
   not((?_sno:Place, ?_tno:Transition ;
        ?_sno:Transition, ?_tno:Place)).
```

Figura 6.15: Un arco debe conectar a nodos de distinto tipo. Es decir, no puede conectar plazas con plazas y transiciones con transiciones. Este predicado identifica la restricción de integridad en el primer parámetro, la red de Petri en la variable `?_net` y el objeto que causó el error en la variable `?_arc`.

traduce un lenguaje unificado que incluye F-Logic, Transaction Logic y HiLog a código Prolog, y este código se interpreta en un motor de inferencia de Prolog llamado XSB[2]. La selección de FLORA-2 como motor de razonamiento está motivada por su licencia LGPL y porque suporta la interpretación de los axiomas de esta ontología [266]. Esto es debido a que FLORA-2 soporta la negación bien fundada mientras que la mayor parte de los razonadores basados en lógica descriptiva, como por ejemplo KAON2[3], sólo soportan la negación estratificada (subconjunto de SWRL [137]) y por ello tienen un soporte limitado de los axiomas definidos para la ontología. Mencionar que FLORA-2 tampoco soporta directamente la axiomática definida, sin embargo, la fachada de redes de Petri añade las reglas para dicho soporte. Por ejemplo, el código de la Figura 6.15 define un axioma para restringir la estructura del grafo, y así evitar que dos nodos del mismo tipo puedan ser adyacentes, a través del predicado `%error(?id, ?red, ?objeto)`.

El razonador FLORA-2 [307] dispone de varios tipos de razonamiento lógico: programación lógica y recuperación de instancias. Por una parte, la programación lógica permite aplicar razonamiento hacia atrás a sentencias en forma de implicación lógica. Por otra parte, la recuperación de instancias es equivalente a comprobar si los hechos tienen o no vinculación con la pregunta, y FLORA-2 realiza muy bien este tipo de razonamiento. La fachada de *reglas de HLPN* define las sentencias para manejar las redes de Petri con este tipo de razonamiento y proporciona un conjunto de predicados predefinidos para manejar tanto las instancias como los tipos de datos básicos (funciones aritméticas y de comparación).

FLORA-2 también incorpora la subsunción como mecanismo de razonamiento. Este tipo de razonamiento se utiliza para inferir que un concepto A subsume a otro B si todas las instancias del concepto B también lo son del concepto A y es muy útil cuando se comprueban las evaluaciones de una red de Petri.

[2]http://xsb.sourceforge.net
[3]http://kaon2.semanticweb.org

Por ejemplo, para comprobar que las marcas tienen el mismo color que la plaza donde se encuentran. La subsunción es necesaria, ya que los tokens son instancias de una clase o subclase de `Carrier` mientras que las plazas lo son de una clase o subclase de `Sort`. Aunque en la ontología una clase Carrier interpreta a una clase `Sort`, la subsunción facilita el razonamiento a nivel de subclases. Por ejemplo, si una plaza con el color *entero* es interpretada por un Carrier con el tipo *numero*, y el valor de los tokens que se insertan en la plaza es del tipo *entero* (subclase de *numero*) entonces la subsunción permite deducir que éstos son tokens válidos para la plaza.

- *Interfaz del razonador*. Se encarga de trasladar los Java *beans* usados para gestionar la ontología. Debido a que cada *bean* está representado mediante una clase en la ontología, esta interfaz traslada cada objeto Java a su correspondiente instancia F-Logic y cada uno de sus atributos a su correspondiente propiedad. Este paso se realiza ejecutando sentencias de la fachada de *reglas de HLPN*. Por ejemplo, la creación de una plaza implica la llamada al predicado de Prolog `%createPlace(?nombre, ?descripcion, ?tipo, ?id)` que ejecuta la sentencia F-Logic y devuelve el identificador de la plaza dentro de la base de conocimiento. Por ello, esta interfaz se encarga de asociar los atributos del objeto Java a las variables de la sentencia de F-Logic. Esta interfaz está desarrollada a partir de la interfaz Java proporcionada por la distribución FLORA-2.

- *Interfaz de persistencia*. Las instancias de la ontología de HLPN se almacenan en una base de datos relacional para hacerlas persistentes. El esquema de la base de datos se basa en los Java *beans* que modelan las clases de la ontología. Estos *beans*, que también se usan para relacionar el modelo Java con el razonador, se anotan mediante el API de persistencia Java que define las relaciones entre el modelo de objetos y el relacional. Esta interfaz también proporciona los métodos para la gestión de dicha persistencia.

- *Interfaz de OPENET*. Aporta los métodos para la gestión, ejecución y persistencia de las HLPN. Por una parte, complementa los métodos de gestión que proporciona la interfaz del razonador y añade nuevas funcionalidades. Por ejemplo, implementa distintas variantes para insertar una red de Petri en el motor:

    - *PNML2J*. Este módulo da soporte a las redes de Petri especificadas en el formato estándar de intercambio de redes de Petri PNML [145]. Por ejemplo, para la carga de un fichero PNML en formato XML se traslada la estructura PNML a los Java *beans* del modelo interno del motor y se traducen dichos *beans* a sentencias F-Logic por medio de la *interfaz del razonador*.

    - *FLOGIC2J*. Este módulo da soporte a las redes de Petri especificadas en el formato de intercambio XML del F-Logic.

    - *OWL2J*. Este módulo da soporte a las redes de Petri especificadas en el formato OWL [84].

Por otra parte, el módulo *gestor de ontologías* se encarga de facilitar la inserción de ontologías del dominio en formato OWL dentro del álgebra de la HLPN. Es una vía alternativa a la inserción de álgebras, ya que éstas pueden insertarse directamente a través de la ontología de HLPN. En este caso, las clases de las ontologías del dominio se cargan como metacarriers del álgebra. Tanto la creación de la taxonomía de la ontología del dominio como la inserción de sus instancias se realiza a través de la *interfaz del razonador*. Esta fachada también maneja la ejecución de HLPN a través del *gestor de ejecución*. Este módulo permite dos tipos de ejecución: guiada o automática. La primera se utiliza para simulaciones o depuración y exige la intervención humana para seleccionar qué transición se va a ejecutar. La segunda, proporciona un planificador que permite ejecutar la red de Petri de forma automática. En ambos casos, la activación de una transición implica que existen las evaluaciones de sus arcos de entrada y salida y que se cumplen sus precondiciones. Es importante mencionar que las funciones que anotan los arcos de la red pueden ser de dos tipos: *(i)* internas (F-Logic) o *(ii)* externas. Las funciones internas se cargan con la ontología dentro del parámetro *expression* de la función. Ya que las funciones anotan los arcos y transiciones de las redes, el motor de redes de Petri es capaz de ejecutarlas de forma automática y así obtener su evaluación. Sin embargo, para dotar a OPENET de más flexibilidad, el motor también permite la creación de funciones sin expressión y cuya ejecución se delega a una entidad externa. En este sentido, el ejecutor soporta dos tipos de funciones externas:

- *Métodos Java.* El planificador realiza la invocación de un método Java e inserta su resultado como una evaluación dentro de la ontología. La invocación de servicios Web también se realiza a través de este procedimiento.

- *Tareas de usuario.* El planificador delega la ejecución de la función al usuario. Éste último es el encargado de introducir la evaluación en el sistema.

La persistencia de las redes de Petri es otro de los puntos importantes de este motor. Esta interfaz almacena los grafos y la ejecución de estas redes en una base de datos relacional. El *gestor de persistencia* facilita los métodos y se encarga de acometer dicha tarea de forma automática o bajo demanda. Cuando un cliente carga una red de Petri, esta fachada crea una instancia en el razonador. Este módulo se encarga de relacionar la instancia del cliente en el razonador con la instancia de persistencia que almacena los datos.

- *Interfaz RMI.* El cliente puede acceder a OPENET vía su interfaz RMI, a través del cual llama los métodos que exporta esta interfaz. Es necesario resaltar que los métodos exportados pertenecen a la *Fachada OPENET* y que los clientes referencian los objetos remotos a través del stub local, el cual actúa como proxy de dichos objetos y es el responsable de redirigir las invocaciones remotas al servidor.

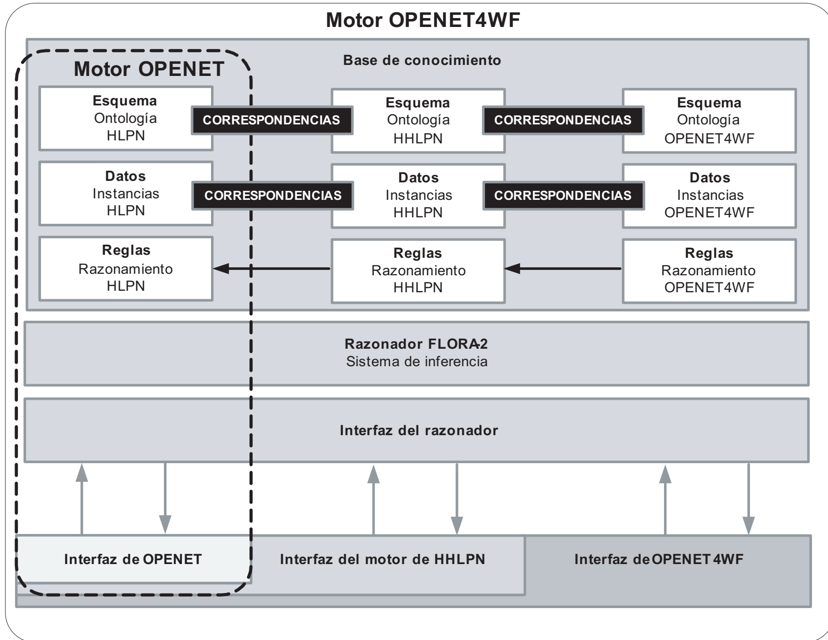


Figura 6.16: Arquitectura del motor de flujos de trabajo. Este motor está construido sobre el motor de redes Petri y le añade una capa para el manejo de redes de Petri jerárquicas, otra capa para el manejo de patrones y una última capa para el manejo de los componentes del metamodelo.

6.5.2. Capa de redes jerárquicas

La Figura 6.16 representa la arquitectura que da soporte a la ejecución de WFs. Como se muestra en la Figura 6.13 este motor extiende a OPENET con dos nuevas pilas encargadas de definir la semántica de *(i)* las redes de Petri jerárquicas y *(ii)* la ontología patrones y los demás componentes del metamodelo de WFs. Las redes de Petri jerárquicas definen la capa intermedia que permite componer redes de Petri complejas a partir de HLPNs más sencillas. Son una estructura formada por HLPNs planas y mecanismos de composición, además del morfismo que asocia la red jerárquica con su representación plana. Esto último es de gran utilidad, ya que no existe una semántica operacional para las redes jerárquicas y, por lo tanto, la toman de las HLPN. De forma más detallada, los principales elementos de la pila de redes jerárquicas son los siguientes:

- *Esquema.* Almacena el conjunto de clases, relaciones, propiedades y axiomas que describen las HLPNs jerárquicas. Al contrario de la ontología de HLPN, no está basada en ningún estándar matemático sino que simplemente recoge los principales mecanismos de composición de redes de Petri existentes en la bibliografía [120]. Básicamente introduce el concepto de fusión y sustitución en

```
%transformHHlpn2Hlpn(?_hhn, ?_fla) :-
   ?_hhn:HHLPN,
   ?_hhn[Pages ->-> ?_pal],
   %createHlpn(?_fla),
   %setProperty(?_fla, HLPN, name, ?_hhn),
   %setProperty(?_fla, HLPN, description, ?_hhn),
   %createMorphism(?_mor),
   %setPropertyList(?_mor, morphism, sourceNets, ?_pal, HLPN),
   %setProperty(?_mor, morphism, targetNet, ?_fla),
   %setProperty(?_hhn, HHLPN, morphism, ?_mor),
   %associateSignature(?_hhn, ?_fla),
   %associateVariables(?_hhn, ?_fla),
   %associateStructure(?_hhn, ?_fla).
```

Figura 6.17: Regla para la transformación de una red jerárquica ?_hhn en una red plana ?_fla. En este caso, el predicado `%transformHHlpn2Hlpn` crea la red plana y el morfismo que relaciona ambas redes.

las HLPN [149] a partir de los cuales se posibilita la construcción de redes más complejas.

- *Datos*. Este módulo almacena las instancias de las HLPNs jerárquicas. Por ejemplo, almacena las páginas (redes de Petri planas/HLPN) que componen una red jerárquica y las instancias de fusiones y sustituciones que definen cómo las páginas de la red se relacionan.

- *Reglas*. El conjunto de reglas definidas para las redes jerárquicas permite *(i)* crear, modificar y borrar redes jerárquicas, *(ii)* crear las copias de las HLPNs de las que una red jerárquica se nutre (páginas) y *(iii)* crear el morfismo que la relaciona con una red plana (HLPN). Este último punto permite crear automáticamente una red aplanada imagen de la red jerárquica donde cada uno de los elementos de la red jerárquica se relaciona con el correspondiente elemento de la red aplanada. Por ejemplo un nodo de una de las páginas que compone la red jerárquica tiene un mapeado en el morfismo que permite identificar su nodo imagen en la red aplanada. La función de mapeado es sobreyectiva, ya que la red jerárquica permite la fusión de un conjunto de nodos. Otro punto interesante es la definición del álgebra que maneja la red aplanada. Ya que una red jerárquica se compone de i HLPNs y cada una de estas redes puede tener un álgebra A_i diferente, el álgebra de la red aplanada resultante será la unión de las A_i álgebras. Por ejemplo, el código representado en la Figura 6.17 permite crear la imagen aplanada de una red jerárquica.

6.5.3. Capa OPENET4WF

La parte más a la derecha de la arquitectura del motor representado en la Figura 6.16 da soporte a la gestión de los conceptos descritos en el metamodelo presentado en esta tesis doctoral [270, 274]. Se trata de la parte encargada de proporcionar los medios para manejar las tareas, métodos, y modelos de control, del dominio y de recursos que describen un WF. La estructura de la base de conocimiento es idéntica a la detallada para las HLPNs o las HLPNs jerárquicas:

- *Esquema.* Almacena el conjunto de clases, relaciones, propiedades y axiomas que describen el metamodelo del metamodelo descrito en el capítulo 4. Además, en este esquema también se describen los patrones de WFs utilizados para construir un modelo de control.

- *Datos.* Este módulo almacena las instancias de los WFs descritos a través de las tareas, los métodos, etc y unidos a través de puentes y refinadores. Por ejemplo, almacenará las tareas que describen una problemática a resolver, los métodos que resuelven dichos problemas, las estructuras organizativas que participan en la ejecución del WF, los modelos de control, etc.

- *Reglas.* El conjunto de reglas de esta capa permite *(i)* crear, modificar y borrar las instancias del metamodelo de WFs y *(ii)* crear la red jerárquica que captura el modelo de ejecución del WF. Este último punto es el más complejo, ya que requiere componer los distintos modelos de control que describen los métodos que resuelven las tareas que forman parte del WF. Una vez seleccionados todos los métodos, la regla representada en la Figura 6.18 se utiliza componer en una única red jerárquica al conjunto de modelos de control que describen operacionalmente estos métodos. El predicado `createExecutionModel` tomará una la lista de modelos de control en la variable `?_lst` y una HHLPN en la variable `?_mod` que contiene el modelo compuesto. Este módulo también contiene al conjunto de reglas que facilitan la creación de los patrones de WFs utilizados para crear un modelo de control. Por ejemplo, la Figura 6.19 muestra parte del código de la regla utilizada para crear un patrón tipo *proceso*.

Finalmente, la *Interfaz del motor OPENET4WF* permite la carga y ejecución de WFs desde Java [267]. Respecto a la carga, posibilita la gestión de cada uno de los componentes de conocimiento del metamodelo descrito en esta tesis. Respecto a la ejecución, integra a las fachadas Java de HLPNs y HLPNs jerárquicas a través de las cuales obtiene el modelo ejecutable de un WF. De forma más detallada, permitirá identificar los componentes de conocimiento, seleccionar/crear los adaptadores para unirlos, y a partir de este modelo traducir en un primer paso el modelo a una red jerárquica y posteriormente a una red plana. La ejecución de la red plana estará directamente a cargo del motor de HLPNs previamente descrito.

```
%createExecutionModel(?_lst, ?_mod) :-
   if (?_lst = [])
   then (?_mod = null)
   else (%createHHlpn(?_mod),
    %addSubstitutions(?_lst, ?_mod),
    %addFusions(?_lst, ?_mod),
    %addPages(?_lst, ?_mod)).

%addSubstitutions([], ?_mod).
%addSubstitutions([?_hhn|?_lst], ?_mod) :-
   ?_mod[substitutions ->-> ?_su1],
   ?_hhn[substitutions ->-> ?_su2],
   append(?_su1, ?_su2, ?_sul)@_prolog(basics),
   addSubstitutions(?_lst, ?_mod).
   ...
```

Figura 6.18: Regla para la unión de un conjunto de redes en una única red jerárquica

6.6. Conclusiones

Existen muchos factores que determinan el éxito de un WMS y que deben se tenidos en cuenta a la hora de seleccionar una herramienta de WFs. Entre ellos, su metamodelo, sus funcionalidades o su infraestructura de ejecución. Este último factor tiene una gran importancia, ya que incide de manera directa tanto en el diseño de los WFs como en la integración de las aplicaciones que van a participar en su ejecución. Por ello, este capítulo se ha centrado en explicar las principales características de la infraestructura desarrollada para dar soporte al metamodelo.

La infraestructura que presentamos está compuesta de cuatro capas, cada una de las cuales da soporte a una parte específica de la ejecución del WF [270, 274]. La primera capa se encarga de la definición de los WFs. Es, por lo tanto, la capa que da soporte al metamodelo descrito en el Capítulo 4 y que permite gestionar cada una de los componentes de un WF. Desde el punto de vista de la ejecución, el principal objetivo de esta capa es proporcionar WFs “completos”, es decir, WFs (pre)definidos a partir de cada una de sus componentes. Aporta por ello las bases para el diseño paramétrico de WFs, ya que un WF se puede ver como el proceso de resolución de un rompecabezas donde cada uno de los componentes muestra una parte de la figura a construir y donde los adaptadores representan el punto de enganche entre dos componentes.

La segunda capa trata la ejecución del WF. A su cargo está desde la creación del modelo de ejecución hasta la interpretación de las HLPNs de cada una de las instancias del WF. Se puede considerar esta capa como el núcleo de la infraestructura, el lugar donde se toman las decisiones que afectan a la ejecución. Dado que el modelo de ejecución es una HLPN, un motor de HLPNs [276] está a cargo de dirigir su ejecución. Es justo comentar que al contrario de otros lenguajes ejecutables de WFs (como por

```
%createProcessPage(?_pro, ?_rel, ?_pag) :-
   %createVariable("x", Context, ?_va0),
   name(?_pro,?_prn)@_prolog(standard),
   %createSort(?_prn, ?_prn, [value], [?_prn], string, ?_prs),
   %addProperty(wfSig, signature, sorts,     ?_prs),
   %addProperty(wfSig, signature, operators, ?_prs),
   %createOperatorApplication(?_prs, [], string, ?_str),
   %createOperatorApplication(precondition,  [?_str, ?_va0], boolean, ?_ot0),
   %createOperatorApplication(boolean_not, [?_ot0], boolean, ?_ot1),
   %createOperatorApplication(postcondition,  [?_str, ?_va0], boolean, ?_ot2),
   %createOperatorApplication(boolean_not, [?_ot2], boolean, ?_ot3),
   append('input_',       ?_prn, ?_inn)@_prolog(basics),
   append('output_',      ?_prn, ?_oun)@_prolog(basics),
   append('enabled_',     ?_prn, ?_enn)@_prolog(basics),
   append('finished_',    ?_prn, ?_fin)@_prolog(basics),
   append('run_process_', ?_prn, ?_rpn)@_prolog(basics),
   append('if_pre_',      ?_prn, ?_ipn)@_prolog(basics),
   append('else_pre_',    ?_prn, ?_epn)@_prolog(basics),
   append('if_pos_',      ?_prn, ?_ion)@_prolog(basics),
   append('else_pos_',    ?_prn, ?_eon)@_prolog(basics),
   %createPlace(?_inn, Context, ?_inp),
   %createPlace(?_oun, Context, ?_oup),
   %createPlace(?_enn, Context, ?_enp),
   %createPlace(?_fin, Context, ?_fip),
   %createTransition(?_rpn, ot_true, ?_rpt),
   %createTransition(?_ipn, ?_ot0, ?_ipt),
   %createTransition(?_epn, ?_ot1, ?_ept),
   %createTransition(?_ion, ?_ot2, ?_iot),
   %createTransition(?_eon, ?_ot3, ?_eot),
   ...
```

Figura 6.19: Regla para la creación de un patrón proceso (I). Esta regla traslada la representación de la Figura 5.2 a una red descrita a través de la estructura de clases de la ontología de HLPNs.

ejemplo BPEL o BPML), el modelo ejecutable de esta capa incluye todos los aspectos que intervienen en la ejecución, desde las actividades a realizar hasta los recursos que las van a resolver. Descontando las diferencias expresivas de cada lenguaje, los lenguajes de WFs no suelen integrar los recursos dentro del modelo de ejecución, de forma que es necesario buscar mecanismos para interceptar las actividades a realizar y asignar externamente los recursos. En cambio, el modelo de ejecución que describimos en este capítulo integra los recursos directamente a través de restricciones en las condiciones de guardia de las transiciones. Así, sólo se posibilitará la ejecución de una transición a aquellos recursos que cumplan su condición de guardia.

Estas dos primeras capas de la infraestructura están soportadas por el motor OPENET4WF, que está a cargo de supervisar la definición de los distintos WFs como del control de su composición y ejecución. El motor OPENET4WF ha sido construido sobre un motor de HLPNs jerárquicas que se utiliza para dar soporte semántico

y operacional a los modelos de control y al modelo de ejecución derivado del WF. Consecuentemente, es posible ejecutar un WF a través del formalismo de las HLPNs asegurando así la no ambigüedad de la ejecución. En este sentido, no es necesario dar un paso extra para traducir el modelo de HLPNs a un lenguaje de ejecución o a un lenguaje de programación, ya que el propio razonador de OPENET4WF se encargará de coordinar la ejecución de las transiciones. Otra ventaja de esta solución es que trabaja directamente al nivel de la ontología de HLPNs, de forma que en cada paso de la ejecución se evalúan los términos que anotan los arcos y las transiciones, se identifican los modos de transición activos y se selecciona el/los modos(s) de transición a disparar. Los resultados de estas evaluaciones se almacenan a su vez en la base de conocimiento de forma que es posible seguir los pasos y el razonamiento del motor en cada paso de la ejecución, y de a su vez verificar el funcionamiento del WF. Consecuentemente, la ejecución de un WF podría intercambiarse entre distintos motores, ya que toda la información dinámica está almacenada en la base de conocimiento y la ontología de HLPNs permite exportarla. Con estas características, el WMS puede implementar una arquitectura distribuida o tolerante a fallos, ya que en cualquier instante la información de ejecución se puede transmitir a otro motor bien sea para mejorar el rendimiento o por una caída del servidor.

La tercera capa implementa un mecanismo de pase de mensajes entre el WMS y los sistemas externos encargados de ejecutar los métodos primitivos del WF. Esta solución permite independizar el WMS de las características de los sistemas externos, y lograr así un bajo acoplamiento entre los clientes y el servidor. El pase de mensajes está basado en el modelo de publicación/suscripción donde el WMS juega el papel de publicador y las aplicaciones externas se suscriben a las noticias dirigidas a un determinado recurso. De esta forma, el WMS simplemente lanza los mensajes dirigidos a un recurso y toda la funcionalidad relacionada con ese mensaje recae en la aplicación externa que implementa dicho recurso. Es fácil deducir que a través de esta estrategia se independiza la implementación del WMS de la de los recursos así como de la solución tecnológica utilizada por dichos recursos.

Precisamente, la cuarta capa se encarga de integrar los sistemas externos en el WMS. Esta capa maneja un conjunto de adaptadores que hacen las funciones de recursos del WMS. Un adaptador representa, por lo tanto, uno o más recursos y está suscrito a los mensajes que se refieren a dichos recursos en el sistema de paso de mensajes. Cuando el WMS envía una comunicación a un recurso, por ejemplo la ejecución de una actividad, el adaptador del recurso recuperará el mensaje y ejecutará la correspondiente funcionalidad. Hacia fuera del WMS, los adaptadores facilitan la integración de las distintas tecnologías en las que están implementadas los sistemas externos, y permiten invocar servicios, servicios web, métodos remotos, etc.

Los siguientes tres capítulos validarán nuestro metamodelo y OPENET4WF. En particular, en Capítulo 7 nos muestra cómo nuestra solución da soporte a la implementación de WFs con un uso intensivo de conocimiento. En el Capítulo 8 podremos apreciar la versatilidad del modelo en el dominio de la Educación y su manejo de WFs complejos que dan lugar a redes de importantes dimensiones. Finalmente, en el Capítulo 9 veremos la forma de extender la capa de patrones de WFs de nuestro

metamodelo para dar soporte a la ejecución de servicios web semánticos expresados en OWL-S.

7

Proceso de presupuestado en la industria del mueble

En este capítulo presentamos una aplicación del marco conceptual descrito a lo largo de esta tesis doctoral a un proceso del ámbito de las aplicaciones industriales. La aplicación resuelve la tarea de presupuestado dentro del dominio de la fabricación de muebles a medida [275] y ha sido desarrollada en conjunción con la empresa Martínez Otero S.L. El resultado, denominado SEEPIM (acrónimo de *Sistema Experto de Elaboración de Presupuestos en la Industria del Mueble*) [268], es un sistema que integra varios flujos de trabajo (WFs) que mejoran tanto la estructura del proceso de presupuestado como los resultados obtenidos. Aunque el nombre de SEEPIM pueda llevar a la confusión hemos preferido mantenerlo por razones históricas, ya que la aplicación inicial, desarrollada en el marco de dos proyectos de I+D industrial[1] se denominaba así. Debe tenerse en cuenta, en cualquier caso, que el sistema no está restringido al presupuestado sino que también abarca otros WFs que le permiten gestionar la planta de fabricación y el control de calidad. En este capítulo se describirá únicamente el WF de presupuestado, ya que es el más representativo de SEEPIM y además de los más interesantes, ya que resuelve muchas de las problemáticas típicas resueltas por los sistemas expertos (planificación, asignación, valoración, etc.). Es por ello un ejemplo representativo del tipo de procesos que pretende resolver nuestro marco conceptual: WFs complejos y que hacen uso intensivo de conocimiento.

7.1. Automatización del presupuestado de muebles

Hoy en día, la industria del mueble se enfrenta al reto de la globalización y a inauditos niveles de competitividad. Los mercados cada vez más competitivos están obligando a las organizaciones a mejorar constantemente sus procesos de negocio. Un proceso clave para llevar a cabo esta mejora está relacionado con el problema de presupuestado de muebles, cuyo principal objetivo es determinar el coste de fabricación de un pedido realizado por un cliente. Conseguir una buena estimación del coste real de producción

[1]El desarrollo de SEEPIM ha tenido el soporte económico de los proyectos “Sistema experto para la elaboración y seguimiento de ofertas en la industria del mueble” (PGIDT01INN35E) y “Sistema de planificación dinámica de la carga de trabajo de la fabricación en la industria del mueble” (PGIDIT04DPI096E)

de un pedido permite un mejor balance de beneficios, un incremento en la cartera de clientes, evitar sobrecargas en la fabricación y el cumplimiento de plazos de entrega.

Existen varias razones para la automatización de esta tarea. Las tareas manuales son una fuente de errores en los procesos de producción e incentivan la falta de comunicación entre las distintas secciones de la compañía. Cada diseñador o experto en fabricación tiene su propio conocimiento y no saca partido del conocimiento de los otros miembros de la organización. En este sentido, el diseño y la fabricación prevista son cuestionables, ya que se pueden basar en información inconsistente y no determinista. Por ello, es necesario incrementar la colaboración y la compartición de información entre áreas funcionales de cara a mejorar la calidad del producto y reducir costes. La automatización del proceso de presupuestado es un importante avance de cara a conseguir estos objetivos. Sin embargo, la automatización del proceso no es suficiente. Los sistemas necesitan soportar un alto nivel de colaboración, sincronizar a las personas, departamentos y demás recursos que participan en el proceso.

Los WFs son uno de los puntos de partida para conseguir estos objetivos. La estructuración de los procesos de negocio como WFs es una de las tendencias emergentes de la industria [66, 305, 139], no sólo por su fácil aceptación e implantación dentro de las empresas, sino porque la tecnología sobre la que se asienta está cada vez más desarrollada [262]. En cualquier caso, implica el desarrollo y adopción de nuevos modelos e infraestructuras de trabajo a las organizaciones [114]. Es más, requiere dar soporte a la gestión de los procesos, reingeniería y capacidades de monitorización [134]. La mayoría de los sistemas de información de las empresas no soportan estas características y su integración es muy limitada [50]. Algunos sistemas muestran una notable habilidad para tratar con los WFs, como los sistemas de gestión de flujos de trabajo (WMSs), sistemas de gestión de procesos de negocio (BPM) o los sistemas de planificación de recursos (ERP). Todos estos sistemas están enfocados a la automatización del proceso, transferencia de datos y compartición de información pero difieren en la aproximación tecnológica [50].

El hecho de diseñar el WF fuerza a las organizaciones a examinar su proceso de negocio de cara a mejorarlo. Es una tarea costosa que consume mucho tiempo pero que permite determinar la estructura del proceso de negocio y de los recursos que participan. En este sentido, el diseño del WF ayuda a proporcionar la información correcta a la persona indicada. Ello también incide en la búsqueda de soluciones para mejorar el desarrollo del producto bien sea a través de técnicas de ingeniería [195] o de guías [36]. Sin embargo, estas tecnologías carecen de técnicas que permitan gestionar el conocimiento. No están pensadas para modelar ni reutilizar el conocimiento ni para refinar dicho conocimiento en un problema en particular. Estos últimos puntos son de vital importancia en dominios empresariales e industriales donde la experiencia y el conocimiento adquirido crean una ventaja respecto a los demás competidores. Las organizaciones necesitan procesos eficientes y una forma de lograrlo es realizando una buena gestión del conocimiento [228]. Esta gestión del conocimiento no se restringe al conocimiento estático como son las reglas o las ontologías [122, 252] sino también al manejo del conocimiento dinámico de la organización [27]. Esto significa que un método que resuelve una tarea puede usarse como una pieza de conocimiento dentro

de la organización. De esta forma, este conocimiento dinámico no se pierde y puede ser reutilizado. En este contexto, el uso de un marco de conocimiento como el desarrollado en esta tesis permite asumir dichas carencias. Como ya hemos comentado, en nuestro marco conceptual se complementan las técnicas orientadas a procesos con las de gestión del conocimiento, con lo que los desarrolladores pueden hacer uso de las ventajas que ambas aproximaciones pueden aportar al modelado de procesos. Por una parte, se establece el conocimiento de cómo realizar el presupuestado y configurar dicho WF en función de las características del pedido y de la situación de la fábrica. Por ejemplo, se puede favorecer la selección de un método para resolver una tarea dependiendo de las características del pedido. Por otro lado, facilita la adquisición, conservación y reutilización de conocimiento experto. La idea es reducir la dependencia de la empresa de la disponibilidad reducida de los expertos y así aprovechar mejor el tiempo de estos últimos.

7.2. Proceso de elaboración de presupuestos

La elaboración de presupuestos constituye un punto crítico en el ámbito de la industria del mueble, ya que de ella depende parte de la generación de negocio y del funcionamiento diario de la empresa. En cuanto a la generación de negocio, el correcto ajuste de un presupuesto es clave de cara a la competitividad. Por ello, es importante averiguar el coste real de producción de un pedido, no sólo porque de esta forma el ajuste de beneficios es más sencillo, sino porque además se asegura que la producción del pedido no ocasionará pérdidas. Respecto al funcionamiento diario de la empresa, la planificación de la producción y la compra de materiales también se ven afectados por esta tarea. Por un lado, la correcta estimación de los tiempos de fabricación pueden evitar sobrecargas de recursos y cargas de producción no asumibles por la empresa. Por otro lado, la correcta estimación del coste de los materiales puede reducir el desperdicio de material, maximizar las capacidades del almacén y maximizar la relación coste/disponibilidad seleccionando los proveedores más adecuados. Por todo esto, desde el punto de vista de la organización, la elaboración de presupuestos es uno de sus pilares básicos.

7.2.1. Estructura

La Figura 7.1 representa el proceso de elaboración de presupuestos. El presupuestado de muebles nace de una necesidad por parte de un cliente. Esta necesidad, recogida por parte de los comerciales de la empresa, se transforma en la especificación de un posible pedido a elaborar (compuesto por cientos, e incluso miles, de unidades de diferentes tipos de muebles). Comienza entonces una fase más técnica de diseño del producto donde se deciden las características de cada elemento del pedido, de los materiales, de los ensamblajes, de los acabados, etc. por parte de la dirección técnica, y se procede a la creación de los planos técnicos de los muebles del pedido a presupuestar. La especificación de estos planos, normalmente a través de herramientas CAD (del inglés *Computer-Aided Design*) y de simulación, da como resultado la definición

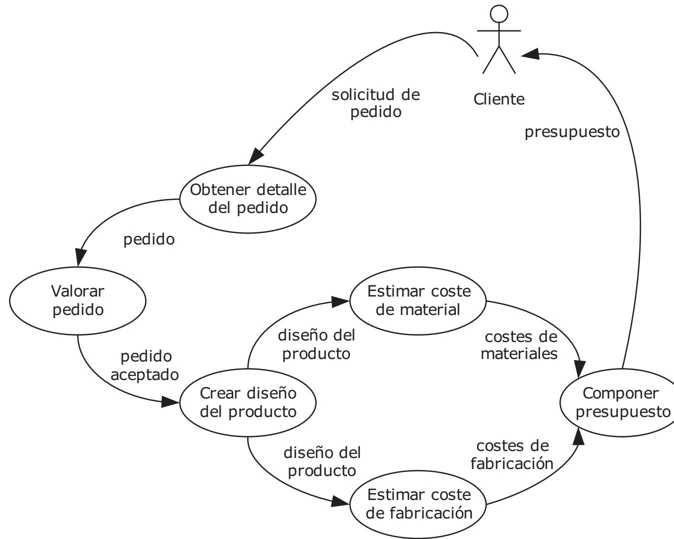


Figura 7.1: Ciclo de vida para la tarea de presupuestado de muebles

de un conjunto de piezas que componen cada mueble (despiece) y de un conjunto de procesos de fabricación para la construcción de cada uno de los muebles del pedido. Una vez aprobados los planos y los procesos de fabricación derivados por parte de los responsables de producto y de fabricación, el proceso sigue adelante con la estimación de los costes de material y de fabricación. La división de compras se encarga de llevar a cabo la estimación del coste de los materiales del pedido. El cálculo de este coste se realiza teniendo en cuenta la información proporcionada por los diseños, del almacén y de los proveedores. Por otra parte, la estimación del coste de fabricación se realiza a partir de los procesos de fabricación definidos y de la carga de trabajo prevista en la planificación de la producción. Una vez obtenidos ambos costes, los responsables de compras y fabricación se encargan de revisar su corrección y entregar los mismos a la dirección técnica para una última revisión. Finalmente, se ajusta el coste final y se establece un margen adecuado de beneficios. El presupuesto queda entonces listo para ser enviado al cliente.

7.2.2. Características

Estimar de forma precisa la fabricación a gran escala de un mueble a medida es una tarea de mucha dificultad. La composición de los muebles es determinante a la hora de estimar el precio del producto. Los muebles a medida se componen de muchos y diferentes tipos de materiales como el metal, la madera o sustratos de madera, plásticos, láminas de melamina, PVC, etc. Esta variedad de materiales complica la evaluación del presupuesto, ya que existen reglas específicas de fabricación y ensamblado en relación con el tipo de material del que se compone el mueble. Además, al ser productos a

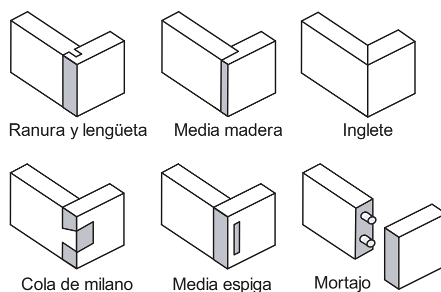


Figura 7.2: Algunas de las uniones de madera más utilizadas

medida no suelen existir referencias ni experiencias previas de fabricación. La fabricación es un proceso complicado y casi cualquier aspecto del diseño afecta directamente a sus costes de fabricación y ensamblaje. En este sentido, la importancia de una estrategia de diseño del producto es incluso mayor, y aplicar metodologías de diseño para la fabricación y el ensamblado (DFMA, del inglés *Design for Manufacturing and Assembly*) [35, 36] es una de las tendencias que permite reducir costes al mismo tiempo que se incrementa la calidad del producto. Las metodologías DFMA consiguen esta mejora definiendo los criterios de cómo fabricar un producto de una forma más efectiva y económica, y consideran para ello información extraída de simulaciones, fabricaciones y ensamblaje. Por ejemplo, la forma en la que dos piezas de madera se unen puede ser un indicativo de la calidad del producto. Una de las medidas de ahorro de costes introducidas por estas metodologías es la reducción y estandarización del número de uniones que pueden usarse en fabricación y la introducción de reglas de cara a seleccionar el tipo de unión más adecuado considerando la dureza, humedad, coste y apariencia final del producto. A título de ejemplo, mostramos en la Figura 7.2 las uniones más típicas entre dos piezas de madera.

El desarrollo de un nuevo producto es más que una actividad creativa. Los diseñadores necesitan tiempo para incubar sus ideas, tener en cuenta los efectos de sus decisiones y adecuarlas a los requerimientos del cliente. Sin embargo, el tiempo para desarrollar un producto es siempre limitado y por ello en bastantes ocasiones los diseños suelen hacerse con prisas. Por esta razón, las organizaciones necesitan formalizar y estructurar sus procedimientos de revisión. Es más, necesitan encontrar una forma de monitorizar el cumplimiento de estos procedimientos y de facilitar un análisis colaborativo del diseño del producto. En este sentido, el seguimiento debe cubrir todas las etapas del ciclo de vida del diseño del producto, coordinando el conocimiento experto entre las tareas y las personas que participan en el proceso. Por ejemplo, la valoración de los diseños de un mueble no puede realizarse con el mismo tipo de conocimiento que el usado en la valoración del coste de fabricación. El primer proceso debe comparar el diseño del producto con las restricciones del diseño conceptual mientras que el segundo debe hacerlo con respecto a criterios de coste y calidad.

Resumiendo, el proceso de elaboración de presupuestos acarrea una serie de ca-

Tabla 7.1: Variables a tener en cuenta en un proceso de fabricación

Variables	*Unidades*
Cantidad de piezas	Entero.
Metros longitudinales	m.
Metros transversales	m.
Metros cuadrados	m^2.
Metros cúbicos	m^3.
Tiempo de preparación	Segundos.
Tiempo de avance	Segundos.
...	...

Tabla 7.2: Variables que intervienen en la elección de un proveedor

Variables	*Unidades*
Precio del material	Precio en euros.
Calidad material	Media de las calidades de las últimas entregas de material por parte del proveedor.
Disponibilidad media	Cantidad de días transcurridos desde la compra del material hasta su recepción.
Capacidad	Cantidad de material que puede asumir el proveedor.
Confianza	Valoración de los encargos llevados por el proveedor.
Fiabilidad	Fiabilidad económica de la empresa proveedora.
Fecha última compra	Fecha.
Calidad última entrega	Medida de la calidad de la última entrega.
Tiempo última entrega	Retardo.
...	...

racterísticas a tener en cuenta a la hora de diseñar SEEPIM:

- Existen muchas fuentes de información, tanto software como humanas.
- Necesidad de coordinación entre las entidades que intervienen en el proceso.
- Conocimiento experto para la selección de rutas de fabricación, realización de operaciones, tipos de uniones, modos de ensamblaje, criterios de calidad, compras, etc.
- Gran volumen de información para caracterizar los muebles. Un mueble se especifica a partir de un plano en formato CAD. A través de este plano, se obtienen el despiece del mueble y los procesos de fabricación (tabla 7.1) para su construcción.
- Gran volumen de información para caracterizar los precios de los materiales. Además de los precios de venta, se necesita caracterizar el material vendido por el proveedor, bien sea midiendo la calidad de los materiales comprados al proveedor a lo largo del tiempo, o a través de los tiempos de entrega. Otros aspectos a tener en cuenta son la confianza asignada al proveedor y los criterios estratégicos. El problema derivado de la gestión de tal cantidad de información se hace más complejo debido al elevado número de proveedores existentes para cada material (tabla 7.2).

- Gran volumen de información para calcular el coste de uso de cada recurso. El cálculo del coste de uso de un recurso depende de muchos aspectos, tales como precio de compra, mantenimiento, programación, etc.

7.3. Estimación de los tiempos de procesado

Uno de los objetivos principales de este desarrollo es reducir la carga de trabajo de los expertos de la empresa. Una de las tareas donde el conocimiento de los expertos era y sigue siendo imprescindible es durante la estimación del tiempo de fabricación en la fase de presupuestado. Esta tarea ocupa gran parte del tiempo disponible de los expertos y por ello no pueden dedicarse a las labores por las cuales han sido contratados. En este sentido, el tiempo dedicado a la tarea de presupuestado va en detrimento de la eficiencia y organización en la planta de producción, ya que, desde la perspectiva de la empresa, la generación de negocio es más importante. La definición de un procedimiento automático que permita estimar los tiempos de fabricación es, por lo tanto, clave para liberar parte de la carga de trabajo de los expertos.

En este dominio, una buena estimación de los tiempos de fabricación cobra una importancia capital: cuanto más exactos sean los tiempos asociados a los recursos que realizan una tarea, mayor será la fiabilidad de la planificación realizada. La fabricación de muebles en este dominio sigue siendo semi-automatizada o incluso manual para algunos procesos de fabricación. Por ello, aunque el tiempo que invierte una máquina en ejecutar su operación está indicado en sus especificaciones funcionales, este tiempo no es realista dado que no tiene en cuenta otros factores que influyen decisivamente en el rendimiento de la máquina (y, por tanto, en el tiempo que invierte en realizar su tarea): la forma de las piezas, la cantidad de piezas que se están tratando en un momento dado, el tipo específico de operación a realizar (por ejemplo, los tipos de corte que es necesario ejecutar sobre una pieza) y el personal asignado para el control de la máquina. Por tanto, no es suficiente con considerar el tiempo indicado en las especificaciones de las máquinas, puesto que al tratarse de pedidos de muebles de cientos e incluso miles de unidades, el desfase entre el tiempo real y el tiempo teórico para tratar una pieza crece de forma significativa a medida que aumenta el número de piezas.

Para resolver esta situación, que impediría encontrar un método fiable para resolver la tarea de planificación, se ha hecho uso de conocimiento experto disponible en la planta de fabricación. Por una parte, se ha analizado el comportamiento de las máquinas a lo largo de miles de horas y para distintos tipos de piezas. Por otra parte, se ha trabajado junto a los expertos de cada una de las secciones de la fábrica para definir tiempos asociados a las máquinas y procesos de fabricación. Como resultado de este estudio se han obtenido una serie de polinomios, cada uno de los cuales estima el tiempo de fabricación asociado a un tipo de máquina y a una determinada operación. Cada polinomio está expresado en función de un conjunto de parámetros que dan cuenta de los factores que intervienen en el rendimiento de la máquina y que son *específicos* para cada una de la máquinas asociadas y al tipo de operación que realiza. En otras palabras, a partir del análisis del funcionamiento de cada máquina durante

un espacio de tiempo significativo se ha obtenido un resultado estadístico para cada uno de los parámetros del polinomio. Por ejemplo, el polinomio correspondiente a la seccionadora de madera es:

$$f(P) = 0{,}1 \cdot x(P) + 3 \cdot y(P)$$

donde las variables tienen el siguiente significado:

- P: esta variable se refiere al pedido del cliente.
- $x(P)$: esta función calcula el número de **metros lineales cortados** . Este valor se obtiene sumando la longitud y anchura del número de tableros de madera a seccionar. Para su cálculo se utiliza la siguiente función donde $UD(p_i)$ representa el número de unidades, $L(p_i)$ la longitud y $A(p_i)$ la anchura de la pieza p_i y todo ello normalizado para un tablero estándar de 2700x200x76:

$$x(P) = n \cdot 2 \cdot 2{,}700 \cdot \left(\frac{\sum_{p_i \in P / madera(p_i)} UD(p_i) \cdot L(p_i) \cdot A(p_i) \cdot E(p_i)}{2700 \cdot 200 \cdot 76} \right)$$

- $y(P)$: esta función calcula el número de **tableros manipulados**. Este valor se obtiene sumando los volúmenes de las piezas a seccionar (donde $E(p_i)$ representa el espesor de la pieza p_i) y se divide por el volumen de un tablero estándar (2700x200x76):

$$y(P) = n \cdot \left(\frac{\sum_{p_i \in P / madera(p_i)} UD(p_i) \cdot L(p_i) \cdot A(p_i) \cdot E(p_i)}{2700 \cdot 200 \cdot 76} \right)$$

Siguiendo una metodología similar, se han identificado los polinomios de las restantes máquinas de la empresa (denominadas centros de coste) y como consecuencia se han obtenido un conjunto de polinomios que dan una medida muy fiable de los tiempos invertidos por cada una de las máquinas y cada proceso de fabricación que realiza. Es importante señalar que este proceso de adquisición ha sido muy arduo y tedioso, ya que un operario ha tenido que tomar nota de los tiempos reales de las máquinas. Asimismo, esta aproximación ha permitido modularizar el proceso de estimación del tiempo de fabricación de un pedido como la suma de los tiempos de fabricación invertidos por los centros de coste por los que pasa. Ello da lugar a que se pueda separar el problema en (sub)problemas más simples y focalizar el esfuerzo de la adquisición de conocimiento en cada polinomio por separado. Ello ha facilitado la validación y refinamiento de los polinomios cuyas estimaciones tienen un mayor error.

7.4. Flujo de trabajo de presupuestado

Como resultado del proceso de diseño, SEEPIM da soporte a un conjunto de WFs para la tarea de presupuestado de muebles [275]. Estos flujos están centrados en el diseño del producto, donde los planos CAD son constantemente evaluados a medida que avanza el proceso. Es necesario mencionar que la solución descrita en esta sección está influenciada por las características estructurales y organizativas de la empresa para la cual se desarrolló SEEPIM. Sin embargo, esta solución es extrapolable a muchos dominios industriales, ya que la mayoría de las empresas se enfrentan a problemas similares cuando abordan esta tarea.

Los WFs que modelan la tarea de presupuestado han sido explícitamente definidos dentro de SEEPIM y son una pieza más de conocimiento con la que razonar. Los restantes componentes de conocimiento se han introducido a través de ontologías asociadas al dominio: reglas de fabricación y ensamblaje, guías prácticas, características de los muebles, etc.

Finalmente, merece la pena mencionar que la mayoría de los métodos utilizados para modelar esta tarea se han fundamentado en la librería de componentes reutilizables definida por la metodología CommonKADS [43]. Esta metodología ofrece un conjunto de modelos asociados a los distintos tipos de problemas existentes en la ingeniería de conocimiento: clasificación, valoración, diagnosis, predicción, configuración, modelado, planificación y asignación. De esta forma, la construcción del modelo de conocimiento se convierte en un proceso de ingeniería consistente en seleccionar o ajustar componentes predefinidos al problema en cuestión. El primer paso a la hora de abordar el problema de elaboración de presupuestos consiste, por lo tanto, en determinar los tipos de problemas a los que se enfrenta. Del análisis del dominio de la aplicación se identificaron tres tipos diferentes de problemas asociados a la elaboración de presupuestos:

- *Problema de valoración.* La evaluación de la viabilidad de llevar a cabo un pedido o de los diseños del producto son ejemplos de este tipo de problema.
- *Problema de asignación.* Este tipo de problema se manifiesta a la hora de asociar precios a los materiales o recursos a las actividades a realizar.
- *Problema de planificación.* Aunque se trate de un presupuesto, es necesario determinar el plan de producción del pedido de cara a determinar posibles fechas de entrega.

7.4.1. Presupuestado

El método que resuelve la tarea de presupuestado está representado en la Figura 7.3. Estas redes seguen con la notación descrita en el Capítulo 3, de forma que cuando una transición está coloreada significa que será sustituida. Para facilitar su lectura, no anotamos todas las transiciones de la red y los métodos complejos se muestran a través

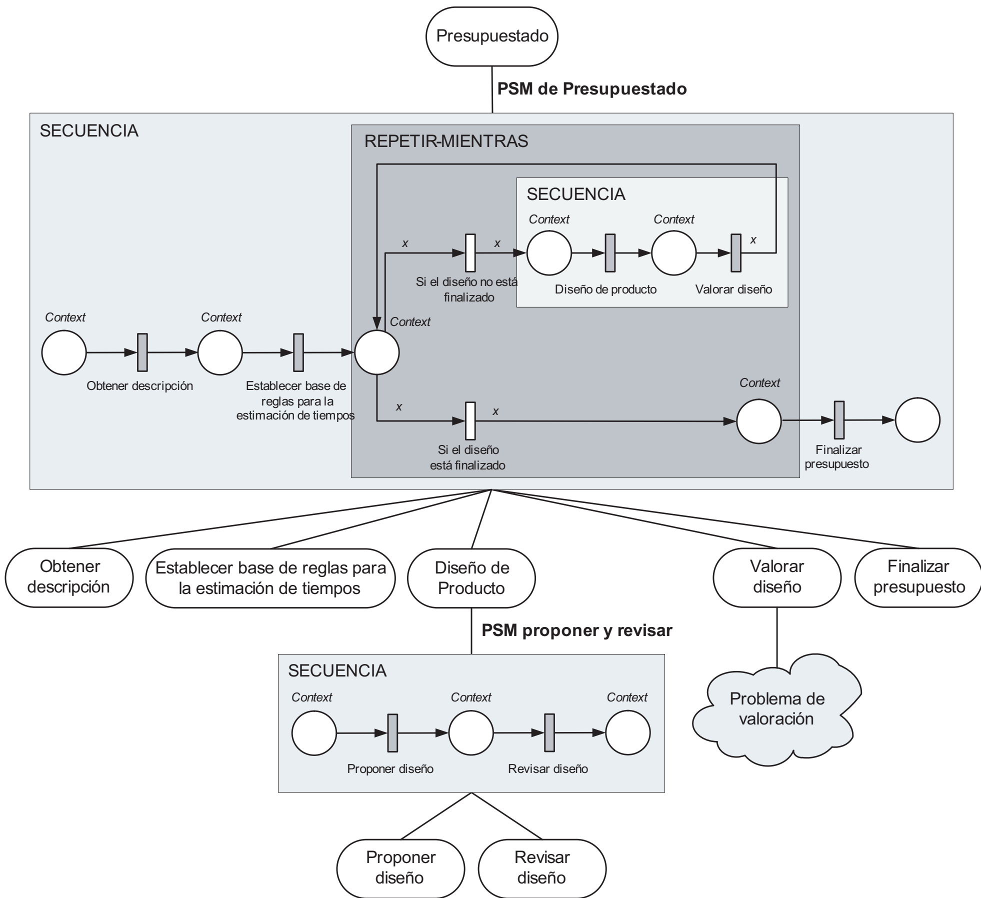


Figura 7.3: Descomposición en tareas del método que resuelve la tarea de presupuestado de muebles

de su árbol de descomposición. Este árbol se lee de arriba hacia abajo y representa las tareas mediante elipses y los métodos mediante rectángulos. Cuando un método resuelve una tarea, existirá un arco entre ambos elementos. Cuando un método se descompone en (sub)tareas, el método tendrá tantos arcos de salida como (sub)tareas. Finalmente, la descripción operacional del método se representa mediante una HLPN. Recordar que en realidad la descripción operacional es una HLPN jerárquica y, por lo tanto, está compuesta por más de una HLPN. Sin embargo, para simplificar dicha representación únicamente se mostrará la estructura de ejecución del proceso y no la red jerárquica completa.

La Figura 7.3 muestra la descomposición de primer nivel de la solución propuesta para este problema. Como se puede apreciar en dicha figura, el método que resuelve el presupuestado obtiene en primera instancia la descripción del pedido a realizar. A

partir de esta descripción inicia un conjunto de iteraciones hasta encontrar el diseño del producto más adecuado para luego finalizar el presupuesto. Cada una de las iteraciones realiza un diseño del producto a partir de la descripción del producto a fabricar. El diseño de un producto es un proceso complicado que difícilmente se realiza en un único paso. Dado que resulta necesario conciliar la componente artística del diseño con los criterios de fabricación y éstos con los criterios de negocio, en cada paso, el director técnico y el departamento comercial se encargan de valorar el resultado del diseño. Esta parte del análisis es compleja, ya que no sólo se supervisan los planos técnicos del producto y los procesos de fabricación necesarios para llevarlos a cabo, sino que también se tienen en cuenta criterios de productividad o estratégicos de la empresa. Por ejemplo, en esta fase se decide si es necesario simplificar partes del mueble o buscar componentes semi-acabados (como son las puestas, cajones, o paneles semi-acabados) que puedan comprarse directamente sin necesidad de fabricarlos y así disminuir la carga de fabricación (aun a expensas del coste).

Se puede apreciar también cómo este método combina los distintos patrones de control hasta llegar a las (sub)tareas que lo componen. La coreografía de este método utiliza dos secuencias y una iteración repetir-mientras. La Figura 7.3 también muestra el método que resuelve la tarea de diseño del producto. En este caso, el WF se modela como un método denominado *proponer y revisar* que es un método genérico utilizado para la satisfacción de restricciones [43, 228]. Este método se emplea para obtener diseños de muebles que sean viables y con un coste adecuado. La obtención de esos diseños se basa en la evaluación y modificación de los requerimientos y restricciones establecidas en el modelo conceptual.

7.4.2. Establecer base de reglas de estimación de tiempos

La Figura 7.4 muestra la resolución de la tarea encargada de establecer la base de conocimiento con las reglas para estimar los tiempos de procesado. La selección de la base de reglas no es trivial y depende del diseño conceptual y del tipo de planificación que se quiere realizar. En ocasiones no es posible obtener un diseño detallado del producto a elaborar y consecuentemente no siempre resulta viable inferir con exactitud las operaciones a realizar. Esta situación es más habitual de lo esperado y se suele producir en las primeras tomas de contacto con los clientes o cuando estos últimos no tienen una idea exacta del producto. En cambio, otras veces el diseño conceptual puede trasladarse a planos CAD y a partir de ellos extraer con exactitud las operaciones a realizar.

El grado de incertidumbre de las dos situaciones anteriores es completamente diferente. En la primera, ni las operaciones ni la mayoría de los parámetros de fabricación están definidos y en caso de estarlo su incertidumbre es elevada. Para estas situaciones es necesario disponer de un conjunto de fórmulas que permitan absorber esta falta de datos: cada fórmula asimilará parte de la incertidumbre a través de los coeficientes de su polinomio y además hará estimaciones a partir de parámetros más globales desligando dicho tiempo de una máquina en particular. Por ejemplo, si el diseño conceptual no dispone de información acerca de si el acabado será manual o

Figura 7.4: Flujo de trabajo del método encargado de seleccionar la base de reglas de estimación de tiempos

automático, no tiene sentido asignar dicha operación a un conjunto de operarios o a un robot de barnizado. En estos casos, la operación de acabado será genérica, no estará ligada a un recurso en particular y aplicará un coeficiente sobre el largo y el ancho de la pieza que permita asumir dicha incertidumbre. Este tipo de reglas se corresponden directamente con las fórmulas vistas en el Apartado 7.3 y adquiridas de los expertos en fabricación y presupuestado de muebles. En cambio, para el segundo caso, los planos aportan todos los datos para aplicar fórmulas muy complejas o herramientas de simulación que permiten obtener resultados próximos al valor real. Para esta situación, se comprobó que a pesar de disponer de más datos, las reglas obtenidas en la fase de adquisición del conocimiento no mejoraban el error medio de las estimaciones. Por este motivo, se optó por aproximar estas estimaciones de forma automática mediante un sistema de aprendizaje de reglas cuya solución está descrita en el Apéndice A.

La tarea *seleccionar base de reglas* aborda esta problemática y permite seleccionar las fórmulas más adecuadas para la estimación de tiempos. Aunque a partir del diseño

conceptual es posible discriminar si se usarán reglas con un alto o bajo nivel de detalle, la selección final recae en la dirección técnica. Es necesario mencionar que la tarea no se limita a la selección entre dos bases de reglas sino también a la versión de la base de reglas. Los pedidos de los clientes suelen evolucionar desde la primera entrevista hasta el diseño final y es preferible utilizar siempre la misma base de cálculo para así mantener un punto de referencia común que permita comparar los distintos presupuestos. Sin embargo, ya que el sistema de aprendizaje actualiza la base de reglas de forma periódica o bajo demanda es necesario fijar la versión de la base de reglas a utilizar.

Sin embargo, en ocasiones el diseño conceptual combina elementos con un alto nivel de detalle con otros poco desarrollados. En esta situación suele ser necesario crear una base de reglas a medida para el presupuesto, es decir, una base de reglas que pueda combinar reglas genéricas con reglas muy precisas. La tarea *aprender base de reglas* activa esta opción a petición del usuario. La construcción de la base de reglas está encuadrada dentro del recuadro marcado como bucle *repetir-hasta* y consiste en asociar la fórmula al proceso de fabricación y máquina que vaya a realizarlo. Como alternativa, el WF también permite elegir la opción de aprender la fórmula con los datos disponibles en la base de datos de productos ya fabricados. El método primitivo que resuelve esta tarea de aprendizaje está detallado en el Apéndice A. Este método se resuelve mediante un modelo de regresión lineal de gran precisión pero que a su vez mantiene la estructura de conocimiento de los polinomios adquiridos de los expertos de la empresa.

7.4.3. Describir detalladamente un pedido

Supóngase, a modo de ejemplo, que el departamento de marketing de la empresa recibe una solicitud de presupuesto que incluye la fabricación de un centenar de muebles para las habitaciones de un hotel. La Figura 7.5 muestra la solución a esta primera fase del presupuestado. Este WF resuelve la tarea denominada *obtener descripción* del método de presupuestado y consiste en fijar los principales requerimientos, restricciones y el diseño conceptual de cada uno de los muebles de los que va a constar el pedido. El primer paso de este método es la introducción del pedido en SEEPIM. Esta descripción incluye el detalle de los muebles a fabricar, es decir, las características superficiales de diseño, del tipo de madera a emplear, de la calidad de la madera, etc. Además de las características de los muebles, también incluye criterios y restricciones de coste y tiempo proporcionados por el propio cliente. Ya que un presupuesto expresa un compromiso firme hacia el cliente, es necesario disponer de una tarea que permita auditar la viabilidad de asumir el pedido a presupuestar. La tarea *valorar pedido* se encarga de realizar esta labor a partir de la descripción del caso. La tarea tiene dos objetivos, *(i)* verificar que se dispone de la suficiente información para elaborar el presupuesto, y *(ii)* verificar que se dispone de tiempo y recursos para asumir su producción. Esta tarea se resuelve adaptando el método clásico de valoración disponible en la librería de componentes de CommonKADS [43].

Siguiendo con el ejemplo y una vez aprobada la creación del presupuesto, el perso-

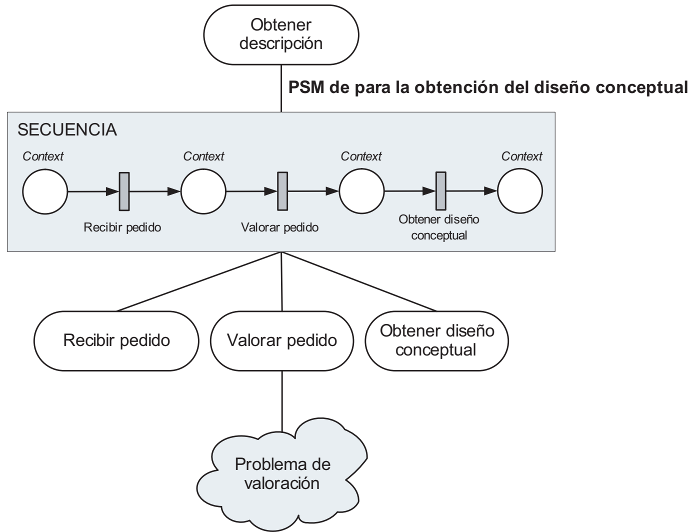


Figura 7.5: Descomposición en tareas del método que resuelve la tarea encargada de crear la descripción del pedido a presupuestar

nal de marketing se encarga de detallar de forma más precisa los muebles aportando incluso los planos de la habitación del hotel. Por ejemplo, una típica habitación podría constar de una cama doble, dos mesillas de noche, una cómoda y un guardarropa. Teniendo en cuenta esta descripción, el departamento técnico decidirá las características principales de los muebles, requerimientos, y restricciones de fabricación y ensamblaje. Por ejemplo, tipos de acabados, uso de material hidrófugo o ignífugo, etc. Al finalizar esta fase, el presupuesto dispone de un modelo conceptual a partir del cual crear los diseños.

7.4.4. Diseño de producto

La actividad principal de la tarea de presupuestado es la definición de un diseño de producto que sea consistente con el diseño conceptual. La Figura 7.6 muestra la descomposición de la actividad *proponer diseño*, que aporta esta funcionalidad. Esta tarea se encuentra dentro de la estructura *repetir-mientras* del WF principal y se ejecutará mientras no se cumplan los criterios de finalización. El primer paso consiste en crear los diseños en formato CAD, los cuales combinan una actividad creativa con el cumplimiento de los requerimientos del diseño conceptual. Además, los diseñadores aplicarán las recomendaciones de la metodología DFMA para así reducir el coste de producción del pedido. Para ello, intentarán usar componentes y materiales estándar, desarrollar un diseño modular, reducir el coste de manejo de piezas y comprobar la robustez del diseño. Por ejemplo, el coste del producto varía en función del tipo de unión usado para ensamblar el mueble: algunas uniones requieren el ensamblaje

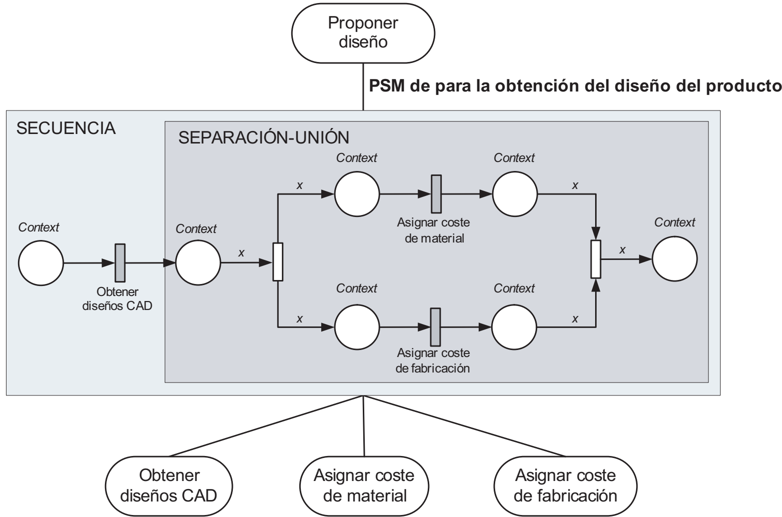


Figura 7.6: Descomposición en tareas del método que resuelve la tarea encargada del diseño del producto

del mueble antes del acabado, incrementando así el coste de fabricación, embalado y envío.

Una vez que el despiece del mueble haya sido obtenido, las dimensiones de cada una de las piezas fijada y los materiales definidos, la tarea *proponer diseño* se hará cargo de inferir los procesos de fabricación a aplicar a cada una de las piezas del mueble. En caso de ser posible, las distintas partes a ensamblar se relacionarán a través de descripciones y conocimiento experto. Este conocimiento se obtiene directamente del catálogo de productos fabricados y es una fuente de gran utilidad para reutilizar diseños previos. Las piezas del mueble se asignan a los procesos de fabricación en base a un diagrama de precedencia desarrollado para minimizar el coste de manipulación, fabricación y ensamblaje. La mayor parte de los procesos de fabricación a aplicar a una pieza se extraen directamente de sus propiedades, por ejemplo, si una pieza tiene un agujero entonces tendrá que ser mecanizada. Estos procesos de fabricación suelen estar estandarizados, (por ejemplo, el número de agujeros de un taladro vertical está estandarizado en función de la longitud de la pieza a taladrar). Los diseñadores también se ocupan de la simulación de cierto procesos a través de herramientas CAM (Computer-Aided Manufacturing). Por ejemplo se suelen emplear herramientas CAM para minimizar el desperdicio de tablero en una seccionadora.

7.4.5. Cálculo del coste de material

En un mercado competitivo como el de la industria del mueble, la compra de materiales a buen precio determina gran parte de la competitividad del producto final. Sin embargo, el precio no es la única variable que interviene en la ecuación. La calidad

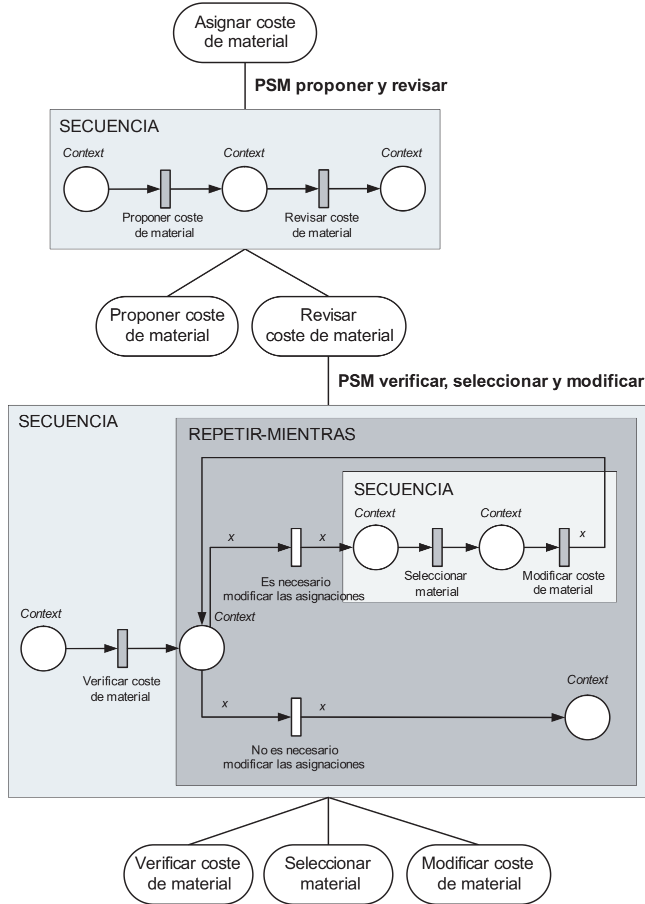


Figura 7.7: Flujo de trabajo del método que resuelve la tarea del cálculo del coste de materiales

del material es otro aspecto a evaluar, ya que la compra de un material de mala calidad puede retrasar o encarecer el producto final. Otro aspecto clave está relacionado con la fiabilidad de los proveedores de material, fiabilidad no sólo en el aspecto de la entrega de material, sino también en otros aspectos como pueden ser su capacidad de llevar a cabo grandes encargos. La tarea *asignar coste de material* se encarga de asociar precios a los materiales empleados en el presupuesto tomando en consideración los aspectos anteriores. La solución propuesta para resolver el problema de asignación se basa en adaptar el método de *asignación* disponible en la librería de componentes de CommonKADS a las necesidades de esta tarea. En este caso, consiste en adaptar la clase general de métodos conocida como *proponer y revisar* a las características de este dominio.

La Figura 7.7 representa la solución a esta tarea. Por una parte, la tarea *proponer*

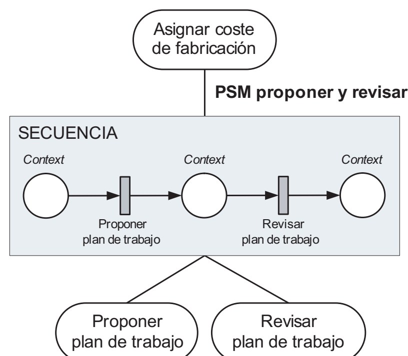


Figura 7.8: Flujo de trabajo del método que resuelve la tarea del cálculo del coste de producción

coste de material se encarga de seleccionar el precio más adecuado para cada material del presupuesto. Este coste se calcula a partir del coste directo del material, del coste logístico, del coste de los aditivos (chapas, barnices, colas, etc.), del coste del material desperdiciado y se fundamenta en un conjunto de reglas/criterios extraídos de los expertos en compras de la empresa. Por el otro, la tarea *revisar coste de material* se encarga de verificar la corrección del precio propuesto para cada material y de modificar dicho precio en caso de no serlo. La revisión se basa en la información proporcionada por los proveedores acerca de sus precios de venta, y revisa entre otros aspectos que el precio ofertado sea actual y siga vigente en el momento de fabricación. Respecto a los proveedores, también se tienen en cuenta aspectos como la calidad de los pedidos anteriores.

7.4.6. Cálculo del coste de fabricación

La Figura 7.8 representa el WF de planificación. Al igual que el cálculo del coste de material descrito en la sección anterior, la solución se basa en la clase general de métodos *proponer y revisar* aunque en este caso aplicada a la generación de planes de trabajo. El método que resuelve la tarea *proponer plan de trabajo* está representada en la Figura 7.9 donde se ejecuta un bucle del tipo *repetir-mientras* mientras el plan de trabajo no ha sido aceptado. El cuerpo del bucle está compuesto por una secuencia de tres tareas. La primera se encarga de fijar el calendario de trabajo de la planta de fabricación. Esta tarea toma el calendario de trabajo establecido para cada recurso a principios de año y permite aplicar cambios a dicho calendario. El calendario puede modificarse de varias formas:

- *Alta o baja de un recurso*. La disponibilidad de algún recurso humano puede verse afectada por una enfermedad, cambio de turno, o simplemente porque ha causado baja en la empresa. La disponibilidad también afecta a las máquinas

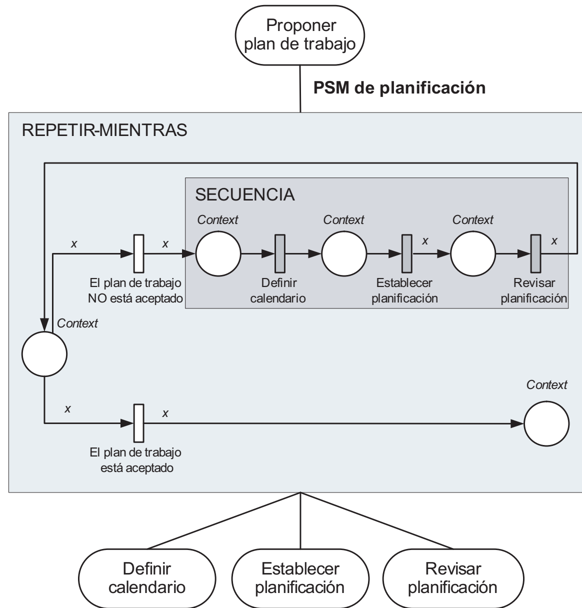


Figura 7.9: Flujo de trabajo del método que resuelve la tarea de generación del plan de trabajo

ya que éstas pueden sufrir averías o simplemente no estar disponibles durante un período de tiempo por motivos de mantenimiento.

- *Horas extra.* Para un determinado pedido, puede ser necesario reforzar un determinado proceso de fabricación, ya que en caso contrario causaría un cuello de botella en la fábrica. En estos casos, una posible solución suele ser la adición de horas extra a los centros de coste.
- *Turno extra de fabricación.* Suele aplicarse cuando la carga de fabricación excede lo asumible en los turnos normales de fabricación y la fecha límite de algún pedido están próxima. Para añadir un turno extra de fabricación se requiere crear una entrada en la agenda de trabajo de cada uno de los recursos que van a trabajar en dicho turno.

La tarea *establecer planificación* es el núcleo de este WF y se encarga de la generación de un plan que asigne el trabajo a los recursos que participan en la construcción del producto. Sin embargo, la planificación no puede verse como un proceso aislado dentro de una fábrica. La fabricación de un producto se ve afectada por los demás productos, es decir, por aquellos en cola para ser fabricados y por los ya presupuestados y pendientes de aprobación. Aunque no siempre es necesaria una precisión absoluta respecto a la fecha de entrega a la hora de realizar un presupuesto, sí es necesario saber si la planta de producción podrá asumir la nueva carga de trabajo en los tiempos requeridos por el cliente. Esta información es crucial a la hora de realizar un

presupuesto, ya que de ella pueden derivarse decisiones como la compra de elementos prefabricados o la subcontratación de la producción (total o parcial) del producto a otra empresa.

El método que resuelve esta tarea está representado en la Figura 7.10. Es necesario comentar que este mismo WF también se utiliza para planificar la producción diaria de la planta de producción en el sistema de planificación. El primer paso consiste en seleccionar los pedidos a fabricar. Por norma general, una empresa dedicada a este tipo de fabricación puede llegar a tener muchos pedidos en cartera, algunos de los cuales son a largo plazo. La planificación de recursos es una tarea computacionalmente muy costosa, por lo que, cuanto mayor sea el número de pedidos que se planifican, mayor será el cálculo a realizar y posiblemente peores serán los resultados obtenidos, ya que el espacio de búsqueda es combinatorio. En este sentido, las fábricas tampoco necesitan saber, con el detalle que proporciona una planificación, el horario de trabajo de un determinado recursos dentro de seis meses. Esta información carece de sentido, ya que en ese lapso de tiempo la configuración tanto de los recursos como de la fábrica puede sufrir muchos cambios. Por ello, la tarea de planificación combina dos tipos de planificación: una detallada que abarca una ventana aproximada de dos semanas y otra menos detallada que cubre a partir de la tercera semana. La selección de los pedidos tampoco es trivial. Existen muchos factores a tener en cuenta. Uno de los más importantes es si ejecutar el pedido completo o partirlo en distintos lanzamientos. Ejecutar un pedido de muchos muebles en un único lanzamiento puede colapsar algún centro de procesado y paralizar los demás a la espera de piezas.

El segundo paso del método consiste en clasificar los pedidos seleccionados en función de su fecha de entrega, de su prioridad y del cliente. Estos criterios ordenarán los pedidos a realizar e influirán en la planificación. Por ejemplo, si el producto a presupuestar es trascendental para la empresa puede incluirse con una prioridad elevada y ser planificado en detalle dentro de la ventana temporal de dos semanas.

El tercer paso consiste en la ejecución de la tarea *planificar trabajo*. Esta tarea se resuelve asignando el trabajo a los recursos a partir del calendario previamente definido. La resolución de este tipo de problema es computacionalmente compleja y requiere del uso de un método de planificación o de asignación de recursos (*scheduling*, en inglés). El sistema SEEPIM proporciona dos algoritmos distintos para acometer esta tarea. Ambas soluciones están descritas en el Apéndice A y están fundamentadas en algoritmos evolutivos. Para más información consultar [281, 282].

Finalmente, a través de la tarea *seleccionar plan*, el responsable de fabricación elegirá el plan más adecuado de entre el conjunto de soluciones propuestas por SEEPIM.

Volviendo al WF que resuelve la tarea *proponer plan de trabajo*, la tarea *revisar planificación* analizará el plan de trabajo obtenido en busca de problemas: reparto del trabajo, sobrecargas, problemas logísticos, disponibilidades de la materia prima, etc. A partir de este análisis, los responsables de fabricación valorarán si se acepta la planificación tal y como está o se solicita una nueva planificación teniendo en cuenta este análisis. Puede parecer extraña la existencia de un proceso de valoración dentro

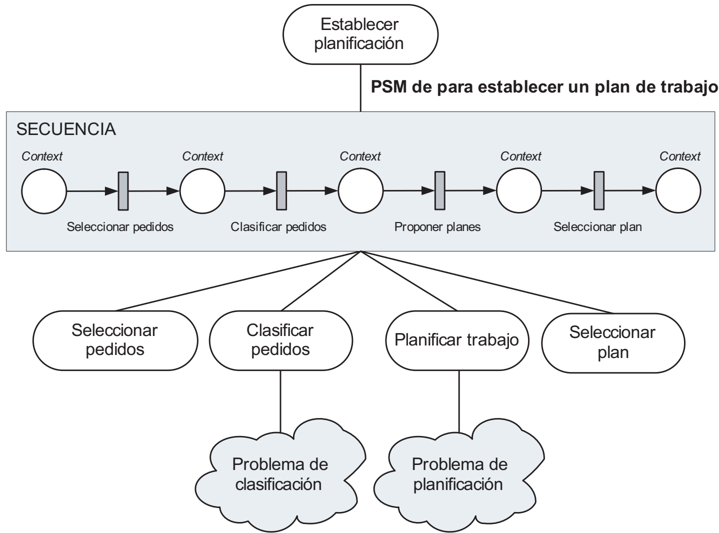


Figura 7.10: Flujo de trabajo del método encargado de establecer el plan de trabajo

de este método habida cuenta que al finalizar la tarea *proponer plan de trabajo* se ejecutará la tarea *revisar plan de trabajo*. Sin embargo, son revisiones a distintos niveles: la que se efectúa en este método atiende a criterios de planificación, es decir, evalúa la planificación resultante y si es necesario cambia la prioridad del producto, establece nuevos turnos de trabajo u horas extra; la que se realiza dentro de la tarea *revisar plan de trabajo* observa el resultado de la planificación y decide si modificar los planos CAD del mueble para mejorar algún aspecto del diseño o por criterios empresariales.

7.5. Arquitectura

La arquitectura propuesta, basada en el paradigma cliente/servidor, se fundamenta tanto en los requerimientos propios del proceso de elaboración de presupuestos como en los específicos del sistema de información de la empresa. Debido a que el proceso de elaboración de presupuestos interactúa activamente con varios sistemas y aplicaciones de la empresa, la arquitectura seleccionada facilita *(i)* la integración del sistema dentro de la Intranet de la organización y *(ii)* la integración con aplicaciones externas. Este requerimiento no afecta únicamente a la integración del sistema con aplicaciones y sistemas externos, sino también a la integración de los recursos de la empresa de cara a minimizar la replicación de información (por ejemplo no replicar la información de usuarios del sistema de gestión del personal), y a la integración de los datos relevantes para el proceso de WF (por ejemplo usar los datos actualizados de la base de datos de almacén).

El modo de interactuar los usuarios con el sistema también influyó en la selección

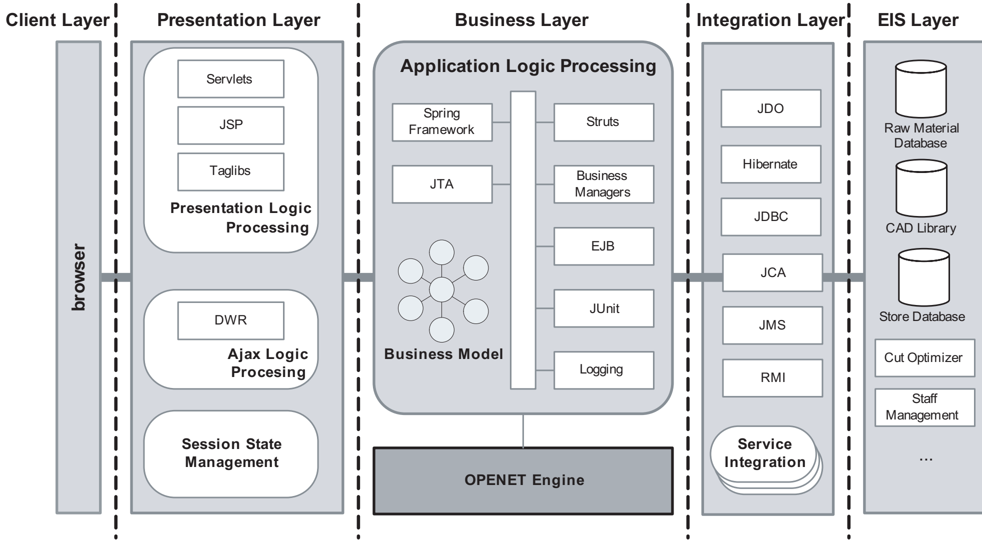


Figura 7.11: Arquitectura software de SEEPIM

de esta arquitectura. Por un lado, aunque inicialmente el sistema no tuviese que dar soporte a un elevado número de usuarios de forma concurrente, de cara a un futuro crecimiento del negocio de la empresa éste debería soportar un número respetable de ellos. Por otro lado en un entorno de flujos de trabajo es importante que la distribución tanto de los participantes como de los servidores sea transparente al usuario (concepto de empresa virtual). Finalmente, facilitar la ubicuidad de los usuarios es un aspecto importante de cara a soportar el acceso desde fuera de la organización de ciertos recursos que participan en el proceso de negocio.

A partir de los requerimientos anteriores, la arquitectura resultante queda reflejada en la Figura 7.11. Para llevar a cabo el desarrollo se optó por la tecnología J2EE[2]. Esta tecnología, además de facilitar la construcción de aplicaciones web, aporta una serie de ventajas que hacen que sea ideal para llevar a cabo este tipo de implementaciones: independencia de la plataforma, madurez, reusabilidad, modularidad, integración con sistemas legacy, etc. Se pueden distinguir los siguientes componentes de la arquitectura:

- *Capa cliente*. Esta capa actúa como interfaz de usuario para acceder al sistema de presupuestos. El uso de esta tecnología facilita el acceso al sistema bien a través de clientes ligeros (por ejemplo navegadores) como de aplicaciones clientes. La Figura 7.12 muestra el acceso al sistema a través de un navegador web. A través de esta capa se accede a las interfaces de usuario, administración y de definición del WF.

[2]http://java.sun.com/javaee

Figura 7.12: Captura de la pantalla de modificación de una especificación de pedido

- *Capa de presentación.* Esta capa se encarga de procesar las solicitudes de los clientes, solicitar la ejecución del correspondiente código de negocio y de crear la presentación al usuario. La capa de presentación se compone de servlets, JavaServer Pages (JSPs), librerías de etiquetas y del marco conceptual DWR[3]

[3]http://directwebremoting.org

para el procesado de acciones Ajax.

- *Capa de negocio*. Esta capa se encarga de procesar las acciones del cliente y está basada en la solución tecnológica de Jakarta Struts [4]. Struts es un marco conceptual OpenSource para implementar aplicaciones web con servlets y JSPs según el patrón Modelo, Vista, Controlador (MVC) [233]. Las aplicaciones Struts tienen, por lo tanto, tres componentes principales:

  - JSPs (Vista).
  - Servlet controller (Controlador).
  - Lógica de negocio (Modelo).

  El funcionamiento es sencillo: el controlador recibe peticiones del cliente (típicamente a través de navegadores web), decide las funciones de la lógica de negocio a ejecutar, y devuelve el resultado al usuario a través de un determinado componente de la capa de vista. La selección de la lógica de negocios está determinada por el motor OPENET4WF. La capa de presentación muestra a los usuarios las acciones que deben realizar en función de la información que les devuelve el motor OPENET4WF. Cuando los usuarios realizan una determinada acción, Struts se comunica con el motor OPENET4WF y éste último le indica el resultado de la acción y el siguiente paso a realizar. La relación entre esta capa y la capa de integración está a cargo del marco conceptual Spring[5]. Este marco proporciona un contenedor de inversión de control a través del cual se externaliza la configuración de la aplicación y con ello se independiza la integración entre las capas de su implementación. Spring también se encarga de gestionar la transaccionalidad a través de JTA[6] (Java Transaction API).

- *Capa de integración*. Esta capa facilita la comunicación de la capa de negocio con las bases de datos, aplicaciones y sistemas heredados de la empresa. La integración de las bases de datos se ha realizado a través de la tecnología Java Data Object[7] (JDO) sobre Hibernate[8]. A través de JDO se accede a la base de datos de presupuestado y a las demás bases de datos de la empresa: base de datos de compras, librería de diseños, materiales, etc. El acceso a otras aplicaciones externas se realiza a través de servicios de integración. Por ejemplo, el sistema se comunica con el software de optimización de corte para averiguar el número de tableros a comprar durante el presupuestado.

- *Capa de sistemas de información de la empresa*. Esta capa contiene a los gestores de bases de datos, aplicaciones y sistemas de la empresa. De entre las bases de datos cabe destacar la librería que almacena los planos CAD que se utilizan para definir la especificación del producto, la base de datos de inventariable de almacén a partir de la cual se deciden qué materiales comprar, la base de

[4] http://struts.apache.org
[5] http://www.springsource.org
[6] http://java.sun.com/javaee/technologies/jta/index.jsp
[7] http://java.sun.com/jdo
[8] https://www.hibernate.org

datos de materiales que contiene las características físicas y técnicas de cada elemento empleado en un mueble o simplemente la base de datos de la fábrica donde los empleados fichan cuando inician o finalizan sus turnos. Al respecto de la elaboración de presupuestos, el sistema también se comunica con aplicaciones tipo CAM (Computer-Aided Manufacturing). Es el caso del optimizador de corte de la seccionadora de madera o el simulador de fabricación del dentro de mecanizado.

De forma resumida, el sistema de presupuestado, además de las funcionalidades que externaliza de OPENET4WF, aporta las siguientes funcionalidades a través de la interfaz gráfica:

- La gestión de los productos.
- La gestión de las especificaciones de producto.
- La gestión de los presupuestos. La Figura 7.13 muestra la captura de pantalla del resumen de un presupuesto. Desde esta pantalla el usuario puede ver los detalles de los materiales, de las operaciones a realizar, del coste de compra, del coste de fabricación o del tiempo de fabricación de cada uno de los muebles del pedido.
- Ver la evolución de los presupuestos realizados a una misma especificación.
- La comparación del coste de los presupuestos con respecto al coste de fabricación.
- El seguimiento de las tareas del sistema.
- La gestión de los usuarios.
- La gestión de los permisos de seguridad.
- La gestión de los sistemas externos.

7.5.1. Integración del sistema de planificación

La arquitectura del sistema de planificación encargado de ejecutar los métodos primitivos relacionados con el cálculo de los costes de fabricación está representada en la Figura 7.14. Debido a la gran carga computacional de este sistema, se externalizó de SEEPIM. Por ello, SEEPIM se comunica con el sistema de planificación a través de la invocación directa de métodos mediante el protocolo RMI[9] (Remote Method Invocation). El núcleo del sistema de planificación está compuesto por tres módulos:

[9] http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

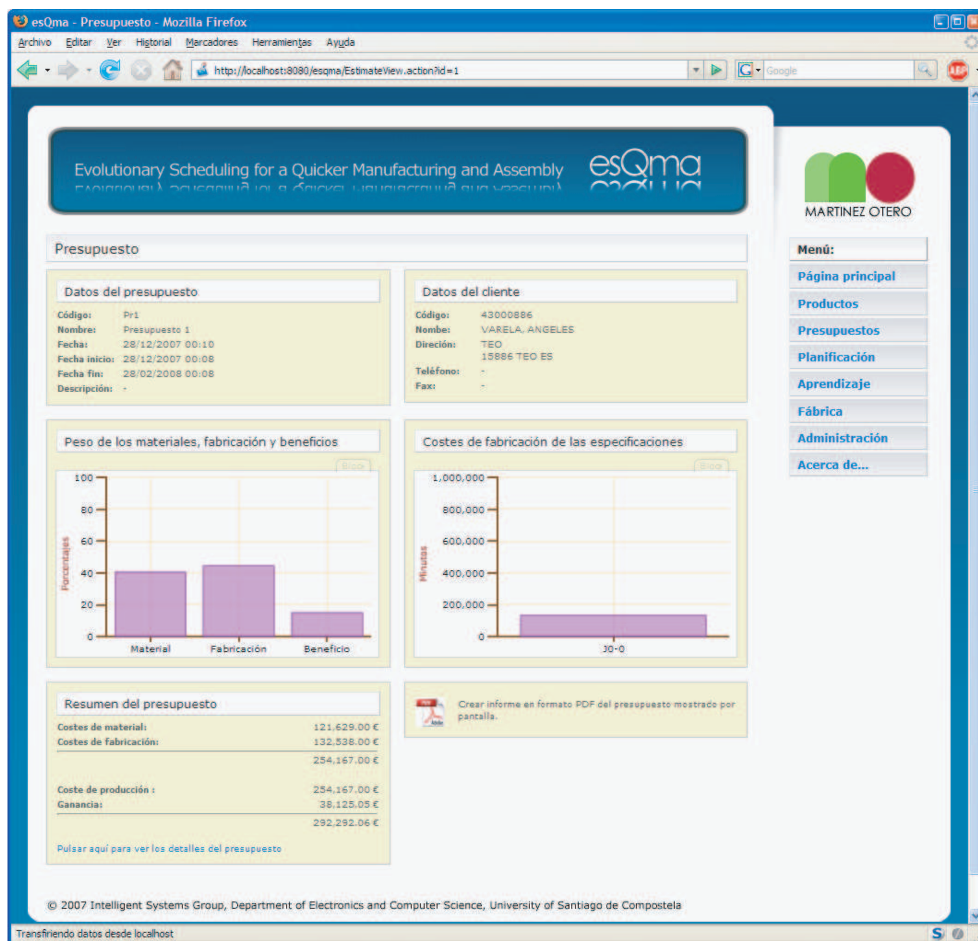


Figura 7.13: Captura de la pantalla del resumen de un presupuesto

- *Planificador*. Este módulo se encarga de planificar los pedidos en un horizonte de tres a doce meses. Específicamente, este módulo calcula las necesidades del pedido a fabricar, estima su tiempo de fabricación y lo añade al plan de trabajo de la fábrica. Es una planificación a grano grueso y no asigna los recursos específicos. Su objetivo es determinar los plazos de fabricación con una determinada capacidad de la fábrica. Cada vez que se añade un pedido, este módulo lo planifica automáticamente dentro de la carga prevista a medio/largo plazo. Este módulo implementa una versión simplificada del algoritmo de planificación visto en este capítulo donde la representación del cromosoma no tiene en cuenta los recursos ý donde únicamente un recurso puede realizar cada tipo de operación.

- *Asignador de recursos*. Este módulo implementa el algoritmo de planificación visto en este capítulo. Esta planificación es a grano fino y, por lo tanto, muy

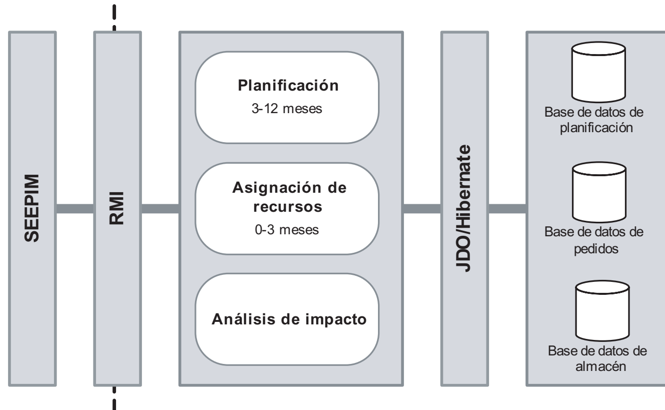


Figura 7.14: Arquitectura del sistema de planificación

costosa. Por esta razón abarca únicamente una ventana temporal de tres meses. Este módulo realiza una planificación diaria o bajo demanda a instancias del WF previamente definido.

- *Análisis de impacto*. Este módulo analiza los planes de fabricación y en caso de encontrar algún problema establece una alerta. Su función principal es analizar si los pedidos pueden fabricarse antes de su fecha de entrega. En caso negativo, calcula las distintas opciones para reconducir el problema. Por ejemplo, calcula el número de horas extra para solucionar un determinado problema, si es necesario establecer un turno extra de fabricación o simplemente si debe de cambiarse la prioridad de algún pedido. Remarcar que este módulo únicamente indica posibles soluciones. Los cambios en las condiciones de planificación son exclusivas de los responsables de fabricación y dirección técnica de la empresa.

El sistema de planificación está integrado dentro de la capa de *sistemas de información de la empresa* de la arquitectura descrita en el apartado 7.5. Además, la interfaz gráfica del sistema de planificación está integrada en SEEPIM. Por ejemplo la Figura 7.15 muestra la visualización del plan de fabricación de un pedido a través de la interfaz web de SEEPIM. A través de la interfaz de SEEPIM se permite proyectar de forma detallada la carga de trabajo en una ventana temporal y en un grano más grueso los restantes pedidos. El sistema dispone de las funcionalidades para indicar en tiempo real la carga de trabajo de un recurso, de un centro de coste o de la propia planta de producción. Además, el sistema también incorpora capacidades para el seguimiento de los planes de fabricación, es decir, para comprobar si el trabajo previsto en el plan ha sido realizado.

De forma resumida, SEEPIM exporta las siguientes funcionalidades del sistema de planificación:

- La gestión de los recursos de la fábrica.

- La gestión de los centros de coste de la fábrica.
- La gestión de los turnos de trabajo.
- La gestión de los rutas de fabricación.
- La gestión del calendario de los recursos.
- La gestión del seguimiento de las operaciones de fabricación.
- La asignación de funcionalidades a los recursos.
- La definición de los polinomios para el cálculo de tiempos.
- Visualización de alertas de fabricación.

Además de estas funcionalidades, el sistema también permite acceder en cualquier instante al plan de trabajo diario de un recurso, de un centro de coste o de un determinado pedido. Estos planes se representan a través de diagramas Gantt y a modo de ejemplo la Figura 7.15 muestra el plan de fabricación de un pedido directamente en el navegador web o externalizada a formato Microsoft Project.

7.5.2. Integración del sistema de aprendizaje de tiempos

La arquitectura del sistema de aprendizaje es muy sencilla y consiste en una capa de interfaces que permiten la invocación del algoritmo de aprendizaje visto en este capítulo. Al igual que el sistema de planificación tiene una gran carga computacional y por ello se integra como una aplicación externa dentro de SEEPIM. La comunicación de OPENET4WF con este sistema se realiza a través del protocolo RMI.

La Figura 7.16 muestra a través de un esquema la dependencia de datos entre todas las partes que integran el sistema SEEPIM. El sistema de aprendizaje es especialmente sensible a esta integración, ya que se nutre principalmente de las estimaciones de tiempos a la hora de realizar los planes de trabajo y de los tiempos reales una vez el producto ha sido fabricado. Las características del producto también juegan un papel importante, ya que permiten clasificar la estimación temporal en función de sus características. Finalmente, el aprendizaje también se alimenta de las incidencias y partes de calidad, ya que permiten descartar mediciones de tiempos erróneas.

Además de integrarse dentro del flujo de presupuestado, el aprendizaje dispone de un WF propio mediante el cual los responsables de fabricación pueden seleccionar las fórmulas a aplicar para la estimación de tiempos de procesado. El WF se inicia periódicamente o bajo petición para así tener siempre actualizadas las fórmulas de tiempos. Estas fórmulas se calculan a partir de la base de conocimiento que propició las fórmulas actuales y los nuevos pedidos fabricados desde entonces. Una vez realizado el proceso de aprendizaje, las fórmulas encargadas de la estimación de tiempos estarán ajustadas y pendientes de un proceso de aprobación.

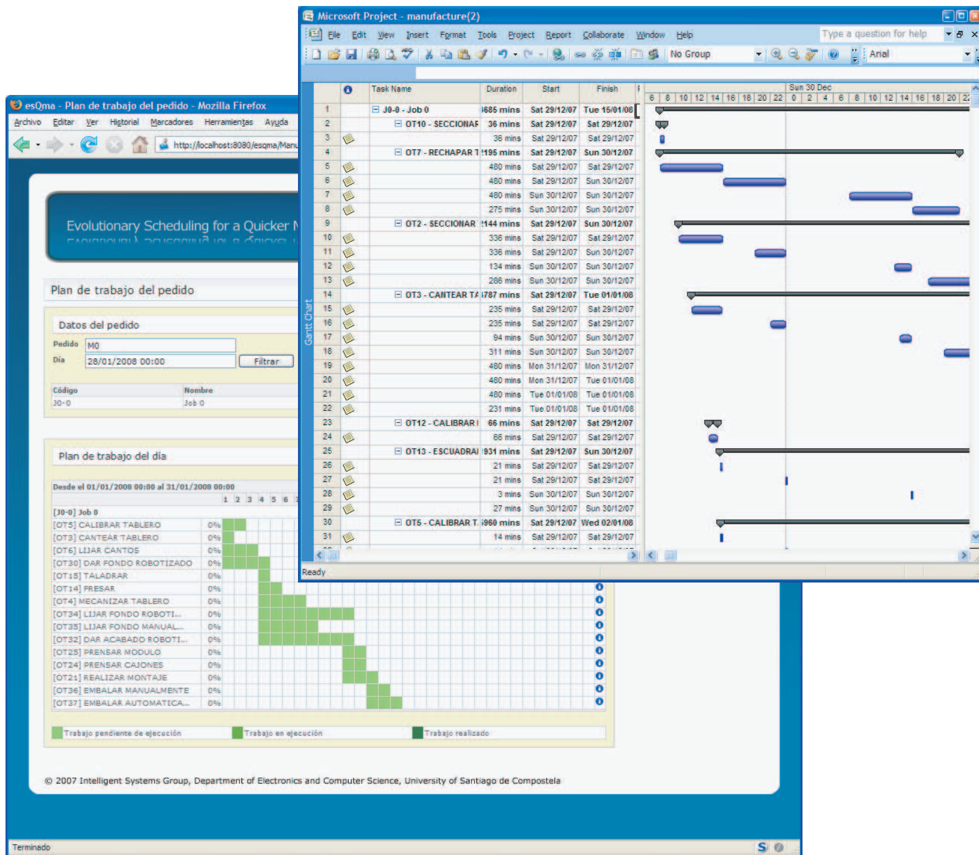


Figura 7.15: Captura de la pantalla de la carga de trabajo asociada a un pedido a través de un diagrama Gantt desde la interfaz web del sistema y exportada a Microsoft Project ™

7.6. Evaluación del sistema

El SBC implementado ha aportado grandes mejoras a la tarea de presupuestado en la empresa del mueble en el que fue implantado. La Tabla 7.3 recoge el error medio de las estimaciones, previas a la implantación de SEEPIM, donde se puede observar como algunos valores superan el 40 %, algo totalmente inaceptable en cualquier industria. Es necesario remarcar que éste no es el error medio de los presupuestos, ya que una empresa con tal ratio de errores no podría sobrevivir en un mercado tan competitivo como lo es la fabricación de muebles a medida. El error en los presupuestos era menor al 15 %, ya que se compensaban las sobre- y sub-estimaciones.

Tras la implantación de SEEPIM se redujo el error a cotas inferiores al 5 % para cada centro de coste. Sin embargo, un análisis exclusivamente numérico no permite

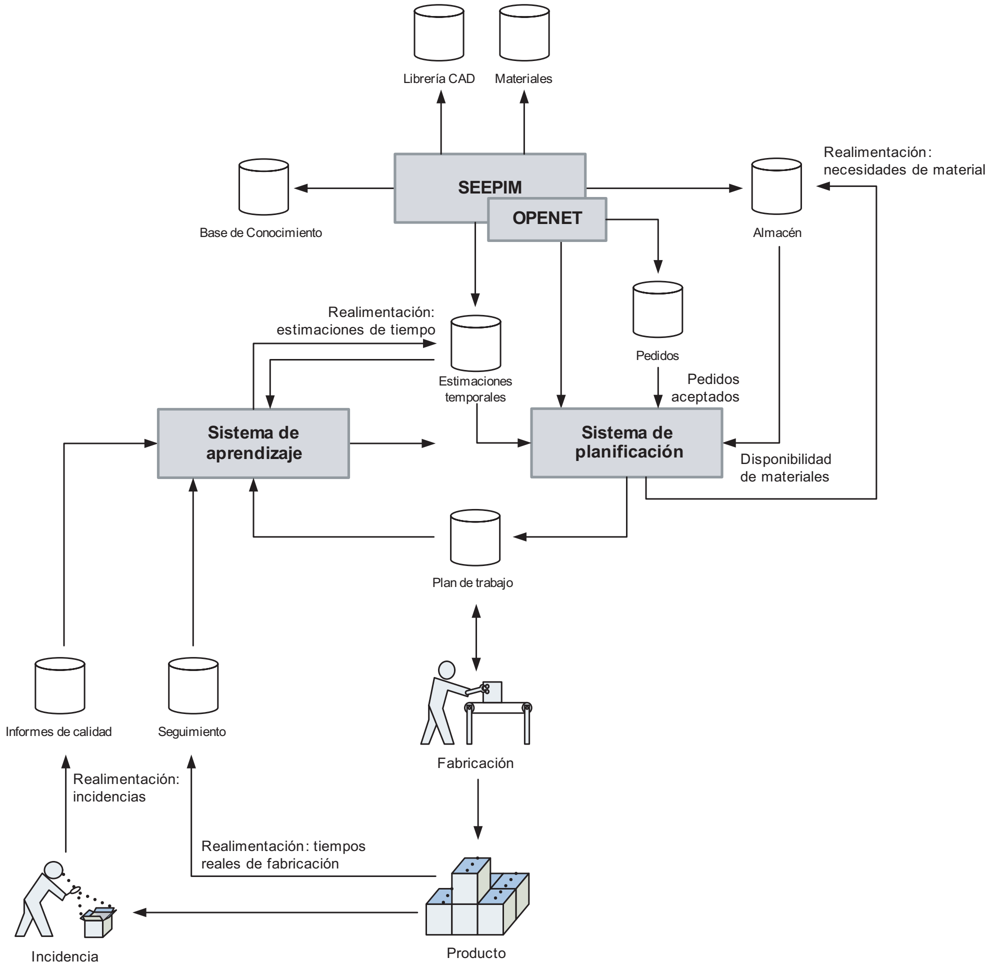


Figura 7.16: Esquema del sistema con la incorporación de los sistemas de planificación y aprendizaje al ciclo productivo de la empresa

apreciar extensión real de la mejora. En este sentido, el procedimiento para alcanzar esta solución y el marco en el que se ha establecido ha sido tan importantes como el modelo descrito en este artículo. Este procedimiento ha llevado a la:

- Reestructuración del proceso de negocio (Modelos de tareas, métodos y control).
- Reestructuración del modelo organizativo (Modelo de recursos).
- Análisis detallado del modelo de negocio y del conocimiento experto necesario para llevarlo a cabo (Modelo de conocimiento).

Tabla 7.3: Error medio en la estimación de tiempo por centro de coste

Centro de coste	Error medio
Seleccion y servir madera	-5,0
Seleccion y servir tablero	-5,0
Seccionadora	-8,0
Seccionadora de madera	-22,0
Plana	+11,4
Escuadradora	+3,4
Canteadora	-14,5
Lijadora Calibradora	-5,5
Lijadora de Cantos	-33,5
Taladro Pequeño	-18,2
Espigadora	-40,2
Tupí	+5,6
Centro de Mecanizado	+7,6
Montaje Intermedio	-26,9
Fondo Manual	-11,2
Acabado Manual	-15,8
Prensa Hidráulica de Cascos	+10,6
Montaje Final	+31,0
Embalaje retráctil	-16,0
Embalaje manual	-22,5

- Normalización del trabajo y de los materiales junto con el empleo de la metodología DFMA para normalizar los procesos de fabricación y ensamblaje de muebles (Modelo de conocimiento).

Esta primera fase de modelado del WF ya reportó beneficios. El principal fue una importante mejora en la estimación de los tiempos de fabricación: un 70 % de la reducción del error de estimación de tiempos se produjo durante el modelado y el 30 % restante tiene una importante dependencia de esta fase, ya que en ella se identificaron las variables que influencian los tiempos de procesado de cada máquina.

El impulso final se dió con la implantación del sistema de aprendizaje automático de tiempos de procesado. El Apéndice A contiene la descripción detallada de este sistema cuyo principal aporte fue la estabilización del error. Aunque la precisión de las fórmulas proporcionadas por los expertos durante el proceso de modelado consiguió una importante mejora, resultó complicado cubrir todo el rango de piezas/casos de entrada para cada uno de los procesos aplicables a una máquina. En tal situación cuando se calcula el tiempo de procesado de una pieza que no encaja dentro del patrón identificado, el tiempo de procesado llegar a tener un importante error. Con la adición del sistema de aprendizaje, se consiguió limitar este problema de forma que, antes de aplicar una fórmula a una operación, el sistema identifica el caso al que pertenece y selecciona su correspondiente fórmula de estimación de tiempos. Para más información acerca del sistema de aprendizaje consultar [189, 188].

7.7. Conclusiones

En este capítulo hemos descrito una aplicación del marco conceptual propuesto en los anteriores apartados de la memoria. El resultado de esta aplicación es el sistema SEEPIM (*Sistema Experto de Elaboración de Presupuestos en la Industria del Mueble*), que está contextualizado en el dominio de la industria del mueble y más concretamente en el problema del presupuestado de muebles a medida.

El problema del presupuestado es de gran trascendencia en cualquier industria, pero tiene incluso una mayor relevancia en la fabricación de muebles a medida, ya que no suelen existir referencias previas acerca de los muebles a fabricar. Este tipo de industria no se dedica a la fabricación de muebles de catálogo donde los tiempos de procesado de cada operación son conocidos de antemano y, por lo tanto, el coste de fabricación sólo depende de los costes de amortización y del margen de beneficios. En la fabricación a medida, los expertos en fabricación se encargan de estimar los tiempos de procesado y a partir de ellos crear el presupuesto. Sin embargo, estas estimaciones suelen ser poco fiables y seguir criterios subjetivo, con lo que es frecuente que exista mucha diferencia entre el coste presupuestado y el coste real del mueble una vez ha sido construido. Cuando el margen de error es a favor de la empresa, el presupuesto corre el riesgo de ser rechazado por el cliente, por considerarlo excesivo, mientras que si es a favor del cliente, su fabricación no resultará rentable para la empresa.

Para reducir el impacto negativo del presupuestado y así mejorar la cartera de negocio de la empresa, se procedió a automatizar esta tarea e integrar entorno a ella varios procesos que hasta entonces se realizaban al margen del presupuestado o de forma manual sin dejar constancia de su realización. Aunque el capítulo se centre en un único WF, se pueden apreciar varias de las ventajas aportadas por el marco conceptual implementado por SEEPIM:

- *Reutilización del conocimiento del dominio.* Gran parte del conocimiento de fabricación relativo a máquinas, operaciones, recursos, etc. se reutiliza en otros WFs. Por ejemplo, un WF se encarga de actualizar el plan de fabricación diario de la planta de producción a partir del trabajo realizado en el día anterior. La base para la nueva planificación es la misma que la utilizada durante el WF de presupuestado.

- *Reutilización de los métodos y componentes de control.* Métodos genéricos como el de valoración se reutilizan en distintos WFs de SEEPIM. Lo mismo sucede con métodos primitivos como el de planificación que se utilizan tanto en el presupuestado como en la planificación diaria.

- *Monitorización del WF.* En cualquier momento se puede acceder al sistema para ver el estado de ejecución en el que se encuentra el WF o la carga de trabajo de cada recurso. Esta información es de suma utilidad y puede utilizarse para monitorizar el comportamiento de los usuarios.

- *Reparto del trabajo.* SEEPIM sigue el reparto especificado en el modelo de recursos de OPENET4WF que en el caso del presupuestado de muebles no tiene

asignaciones individuales para así no bloquear el presupuesto ante la ausencia de una determinada persona.

Desde el punto de vista de la propia tarea de presupuestado, la automatización del proceso y su modelado y ejecución a través de un WF ha tenido beneficios tangibles como:

- *Relación coste presupuestado vs. coste real.* La automatización de la tarea ha reducido el desfase entre el coste presupuestado y el coste real. Cuando el presupuestos se especifica a partir de un diseño detallado, el error respecto al coste real está por debajo del 10 %, el cual es perfectamente asumible teniendo en cuenta el margen de beneficios.
- *Agilidad.* Si descontamos el tiempo necesario para la elaboración de un diseño técnico detallado, la elaboración de un presupuesto pasó de ser cuestión de días a horas.
- *Mejores estimaciones.* La estimación de tiempos es uno de los principales problemas del procedimiento original de presupuestado. En parte esto es debido a que cada uno de los expertos estimaba los tiempos atendiendo a su experiencia (conocimiento), pero también porque la fabricación de muebles a medida tiene características particulares que hacen que estimaciones precisas sean particularmente complicadas de obtener. Para reducir el error de estas estimaciones se automatizó esta tarea y los resultados de esta actuación pueden consultarse en el Apéndice A y en [189, 188].
- *Mejor planificación.* En la mayoría de las industrias la carga de trabajo de la planta de fabricación también afecta al cálculo de los presupuestos. Por ello, es importante disponer de una herramienta que permita integrar los pedidos a presupuestar dentro del plan de fabricación y así poder determinar su incidencia. Por este motivo, SEEPIM integra un planificador que optimiza una ventana de trabajo de dos semanas. Los resultados de esta actuación pueden consultarse en [281, 282] y en el Apéndice B.

Y otros más intangibles como:

- *Reestructuración del proceso de presupuestado.* Antes de la automatización del WF, el proceso no tenía una estructura fija. Además, todo el trabajo recaía en la dirección técnica que ante tal sobrecarga no podía analizar en profundidad los presupuestos. Ahora, el proceso de presupuestado se ha repartido más homogéneamente entre las distintas divisiones de la empresa e incluye varios ciclos de revisión que garantizan una mayor calidad del presupuesto: los costes de material se aprueban en la división de compras, el plan de trabajo en la división de fabricación y cuando el presupuesto llega a la dirección técnica y comercial sólo es necesario analizar si el presupuesto cumple con los requisitos del cliente y con los criterios estratégicos de la empresa.

- *Reestructuración organizativa.* A la vez que se reestructuró el proceso de negocio, también se cambió la estructura organizativa de la empresa. Con la nueva estructura y la adición de nuevos perfiles, cada participante tiene claro cuáles son sus deberes y responsabilidades.
- *Estandarización de los procesos de fabricación.* El diseño del producto ya no es una tarea exclusivamente *artística* y ahora tiene en cuenta estándares de fabricación a la hora de diseñar los planos técnicos. Por ejemplo, se estandarizó en número de taladros a perforar verticalmente en una pieza en función de su longitud, de tal forma que los operarios no tienen que cambiar constantemente la configuración de la brocas de la taladradora vertical con cada cambio de pieza.
- *Estandarización de los materiales y procesos de fabricación.* La base de datos original organizaba los materiales por nombres introducidos desde la división comercial, de diseño del producto, de compras o de almacén. Como resultado, un mismo material se encontraba replicado múltiples veces en la base de datos y, lo que es peor, su referencia en los planos CAD no coincidía con la del almacén. Para evitar estos problemas la empresa implantó un sistema de codificación de materiales no ambiguo y estableció un conjunto de directrices que definen el protocolo a seguir para añadir un nuevo material a la base de datos. En este mismo sentido, las directrices promueven el uso y compra de materiales con tamaño estándar, ya que son más baratos.
- *Estandarización del proceso de obtención de tiempos en la planta de producción a través de lectores de barras.* Uno de los principales problemas con el que nos encontramos a la hora de proceder con la estimación de los tiempos de fabricación es que los tiempos de fichaje de los operarios no se correspondían con el trabajo real y *no eran fiables*. Ello se debía a que tras cada operación de procesado eran los propios operarios los responsables de registrar manualmente su trabajo. Como consecuencia, se imputaban horas a proyectos equivocados, el número de horas imputadas no se correspondía con las reales, y las operaciones a las que se asignaba el tiempo no se correspondían con las especificadas en los diseños CAD. Para evitar estos problemas se procedió a implantar un sistema de fichaje alternativo basado en lectores de códigos de barras. El fichaje dejó de ser responsabilidad de los operarios y pasó a estar a cargo de los responsables de cada sección que al asignar el trabajo se encargan de imputar las horas mediante dichos lectores.
- *Explicitación del conocimiento para el cálculo del coste de materiales.* La explicitación de este conocimiento ha permitido automatizar el proceso de cálculo del coste de material con la consiguiente reducción de la carga de trabajo del departamento de compras. Aunque no es una tarea demasiado complicada, el cálculo de la cantidad de material necesario para fabricar un producto no es una tarea que se resuelve de forma directa. Para ello se debe tener en cuenta la fecha en la que se va a fabricar el producto, la disponibilidad en almacén, la cantidad a comprar, reducir el desperdicio, los costes de almacenamiento, la información de los proveedores, etc.

- *Explicitación del conocimiento para la estimación de los tiempos de fabricación.* Al explicitar este conocimiento muchos de los responsables de fabricación que antes dedicaban gran parte de su tiempo al presupuestado pueden ahora centrar su atención en el control y mejor de los procesos productivos de la fábrica.

- *Explicitación del conocimiento para acometer la planificación del trabajo.* Originalmente la empresa realizaba la planificación del trabajo manualmente. Teniendo en cuenta la complejidad de esta tarea, la planificación era de muy alto nivel y no detallaba las operaciones a llevar a cabo ni el momento en el que se debían realizar. Esta decisión estaba a cargo de los responsable de fabricación que se encargaban de repartir el trabajo en fábrica en función de sus propios criterios. Para mejorar este proceso, se adquirió el conocimiento para planificar la producción de forma automática con los objetivos de minimizar el tiempo de fabricación y repartir la carga de trabajo entre los distintos recursos.

8

Motor de ejecución de unidades de aprendizaje basadas en IMS Learning Design

En este capítulo se presenta una segunda aplicación de nuestro marco conceptual de flujos de trabajo (WFs) [235]. Esta aplicación se desarrolla en el dominio de la educación y permite apreciar la versatilidad del metamodelo propuesto y del motor OPENET LD que lo implementa. Este desarrollo se centra en la creación de un conjunto de WFs [279] para la ejecución de unidades de aprendizaje (UoLs, del ingés *Units of Learning*) en Educación descritas siguiendo la especificación IMS[1] Learning Design (IMS LD). El motor de ejecución de UoLs [280] ha sido desarrollado en el marco de dos proyectos del plan AVANZA[2] y es accesible desde la siguiente dirección: http://www.gsi.dec.usc.es/hlpno/ims-ld-engine.html.

8.1. Unidades de aprendizaje

La necesidad de manejar recursos reutilizables ha llevado al desarrollo de diversas especificaciones de metadatos con el objetivo de representar contenidos de aprendizaje, recursos educacionales y metodologías de diseño de aprendizaje. Las especificaciones para el diseño de aprendizaje, conocidas como lenguajes de modelado educativo (EMLs, del inglés *Educational Modelling Languages*), son modelos de agregación e información semántica que describen tanto el contenido como las actividades educativas. Los elementos en un EML se organizan en UoLs con el objeto de permitir su reutilización e interoperatividad entre herramientas de aprendizaje [207].

Estos lenguajes se utilizan para describir desde una perspectiva pedagógica el diseño de un proceso de enseñanza/aprendizaje. En este sentido, todo profesional de la educación se ha enfrentado en algún momento a la elaboración de un programa de una determinada actividad formativa: asignatura, curso, presentación, seminario,

[1]IMS Global Learning Consortium es una organización de carácter global y sin ánimo de lucro que busca la aplicación de la tecnología a la educación. En http://www.imsglobal.org/ se puede encontrar el sitio web oficial.

[2]El desarrollo de motor de unidades de aprendizaje ha tenido el soporte económico de los proyectos "SUMA elearning multimodal y adaptativo " (TSI-020301-2008-9) y "FLEXO: Desarrollo de aprendizaje adaptativo y accesible en sistemas de código abierto " (TSI-020301-2008-19)

etc. Se plantean entonces aspectos como los objetivos de aprendizaje, el contenido de la actividad formativa, el método docente a aplicar, etc. Los EMLs proporcionan este medio en el que representar el programa o plan docente, permitiendo al educador dar a conocer su programa a los demás implicados (profesores, alumnos, etc.) y, en determinados casos, también proporcionan un marco para la reutilización. De entre todos los EMLs, la especificación IMS LD [119] ha emergido como el estándar *de facto* para la representación de los diseños del proceso de enseñanza/aprendizaje.

El IMS LD surge para facilitar el diseño, comunicación y reutilización de procesos de enseñanza/aprendizaje. Es principalmente una especificación centrada en formación en línea (e-Learning) que permite modelar programas. La especificación IMS LD permite representar una gran variedad de modelos pedagógicos, y adaptar sus recursos de una manera completamente flexible [48]. Mediante la descripción de los diferentes actores, roles, actividades, entornos, método docentes, propiedades, condiciones y notificaciones, se pueden transformar las planificaciones de un aula en una UoL. El concepto de diseño de aprendizaje (LD, del inglés *Learning Design*) se refiere a esta definición e IMS lo define como: un modelo pedagógico que describe varias actividades para un grupo de personas con el propósito de hacer que éstas alcancen un objetivo de aprendizaje determinado, en el contexto de un dominio de conocimiento específico.

En este capítulo se presenta un nuevo motor de UoLs, llamado OPENET LD [280], construido sobre el motor OPENET4WF [276]. Además de las ventajas inherentes del modelo de WFs descrito en esta tesis doctoral, el nuevo motor presenta dos características que lo diferencias de otras implementaciones disponibles en la bibliografía. Por un lado, aporta una base formal a la descripción de las UoLs [279]. Cada UoL se describe como una HLPN y tiene, por lo tanto, una semántica operacional no ambigua. Por otro lado, es uno de los pocos motores que permite ejecutar UoLs de nivel B.

8.2. IMS Learning Design

En 1997 la Open University of Netherlands (OUNL) decidió hacer accesibles sus cursos en Internet. Los cursos existentes empleaban una variedad de enfoques pedagógicos y, por tanto, la Universidad los clasificó y empezó a implementar unas plantillas representativas que podían dar soporte a todas estas categorías pedagogicas. Rápidamente se hizo evidente que todos los profesores tenían su propia visión pedagógica, y que se necesitaban casi tantas plantillas como profesores. Sin embargo, se observó que aunque había muchas descripciones pedagógicas de los cursos, en la práctica todas consistían en combinaciones de tres elementos básicos: recursos educativos, múltiples personas actuando en varios roles, y actividades pedagógicas. Para facilitar la descripción de estos cursos, la OUNL creó un EML que permite usar estos tres elementos para especificar la estructura de una UoL en un documento XML [156].

Conforme a la Figura 8.1, el EML de la OUNL utiliza como principio que independientemente de la técnica pedagógica aplicada, una persona (normalmente un alumno o un profesor) juega uno o varios roles y con ellos realiza ciertas actividades en un

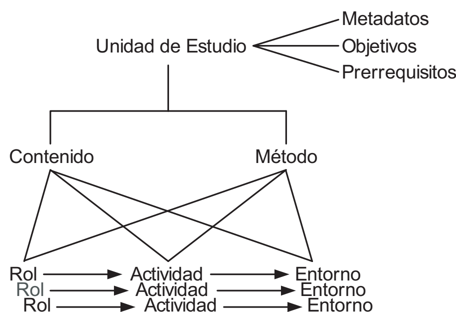


Figura 8.1: El concepto de unidades de estudio en el EML de la OUNL [156]

entorno con el propósito de alcanzar determinados objetivos de aprendizaje. Estas actividades se definen en función a ciertos prerrequisitos y se organizan de acuerdo a una secuencia descrita en un método representando un *flujo de aprendizaje*.

La especificación IMS LD, preparada por el grupo de trabajo IMS/LDWG, es una integración del EML desarrollado por la OUNL [156] con otras especificaciones existentes para el intercambio e interoperabilidad de material de aprendizaje electrónico, . Puede considerarse esta especificación como una capa integradora en la que se incluyen otras muchas estándares:

- *IMS Content Packaging*, permite integrar el IMS LD en la creación de una unidad de aprendizaje.
- *IMS/LOM Metadata*, permite integrar metadatos en diversas estructuras de IMS LD.
- *IMS Question and Test Interoperability*, permite integrar cuestionarios y tests como recursos en actividades de aprendizaje.
- *IMS Reusable Competence Definition*, pueden ser integrados como recursos asociados a los objetivos de aprendizaje y prerrequisitos.
- *IMS Learner Information Package*, permite integrar la información de estudiantes en la estructura de IMS LD.
- *SCORM*, permite integrar contenidos SCORM[3] en IMS LD.
- *IMS Enterprise*, permite asociar participantes (roles) en la creación de instancias de diseño de aprendizaje.

[3]SCORM (del inglés *Sharable Content Object Reference Model*) es una especificación que permite crear objetos pedagógicos estructurados. Los sistemas de gestión de contenidos web originales usaban formatos propietarios para los contenidos que distribuían. Como resultado, no era posible el intercambio de tales contenidos. Con SCORM se hace posible el crear contenidos que puedan importarse dentro de sistemas de gestión de aprendizaje diferentes, siempre que estos soporten la norma SCORM. En http://www.scormsoft.com/scorm se puede encontrar el sitio web con la especificación SCORM.

- *IMS Simple Sequencing*, permite secuenciar objetos de aprendizaje y puede ser incluida como instancias de documentos IMS LD.

En general, cada especificación IMS se encuentra detallada *de modo informal* en tres documentos:

- *Best Practice Guide*, que contiene recomendaciones sobre la forma de uso de los conceptos incluyendo ejemplos y la relación con otras especificaciones.
- *Information Model*, que contiene una descripción detallada e independiente del formato físico de representación de cada elemento que forma parte del flujo o diseño de aprendizaje.
- *XML Binding*, en el que se describen los detalles técnicos relativos a la representación del IMS LD en XML.

El documento *information model* del IMS LD, a su vez, se estructura en tres partes: *(i)* un modelo conceptual, que define el vocabulario y las relaciones funcionales entre los conceptos del LD; *(ii)* un modelo de información, que describe de un modo informal (lenguaje natural) la semántica de los conceptos y las relaciones entre ellos introducidos en el modelo conceptual; y *(iii)* un modelo de comportamiento, que especifica las restricciones impuestas al sistema software cuando un LD determinado se encuentra en tiempo de ejecución. En otras palabras, el modelo de comportamiento detalla la semántica de la especificación IMS LD durante la fase de ejecución y constituye la base del modelo formal que se presentará en el siguiente apartado.

Tanto el modelos de información y de comportamiento del IMS LD utilizan la metáfora del teatro para ayudar a entender la estructuración y ejecución de las UoLs. Así, una serie de actores participan en una representación en la que cada uno de ellos puede asumir un número de roles en diferentes momentos de la representación (es decir, en varios actos). Del mismo modo, un estudiante puede asumir diferentes roles en diferentes etapas del proceso de aprendizaje. Al final de cada acto, la acción se detiene, todos los estudiantes se sincronizan y todo puede volver a empezar.

En la especificación IMS LD existe un conjunto de elementos que describen la forma en la que se ejecutan las actividades educativas; es decir, estos elementos definen el WF que coordina la ejecución de las actividades a llevar a cabo por los participantes en el curso o unidad de aprendizaje, tal y como muestra el ejemplo de la Figura 8.2. Así:

- Un método está compuesto por un conjunto de representaciones que se ejecutan en paralelo. Las representaciones modelan, por ejemplo, unidades de aprendizaje en las que los estudiantes se dividen en grupos que se gestionan de forma independiente unos de otros.
- Cada representación consta de un conjunto de actos que se ejecutan en secuencia y que se pueden ver como etapas o módulos dentro de un curso. En este sentido,

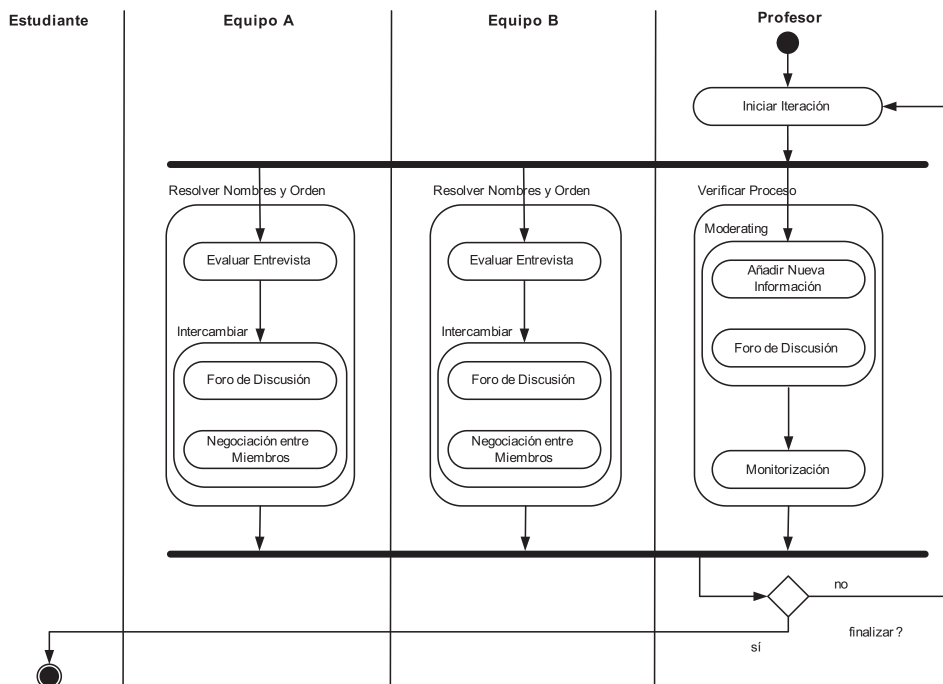


Figura 8.2: Diagrama de actividades del escenario real de aprendizaje en astronomía

los estudiantes asociados a una representación realizan diferentes actos en los que van cubriendo los objetivos pedagógicos.

- En cada acto todos los usuarios que juegan un cierto rol (Grupo A/B y Profesor en el ejemplo de la Figura 8.2) ejecutan un conjunto de actividades, de modo que dichas actividades se realizan en paralelo por cada uno de los roles.
- Para realizar cada actividad los estudiantes y profesores usan una serie de recursos y de servicios educativos con los que alcanzar los objetivos pedagógicos asociados a dicha actividad.

8.2.1. Niveles de implementación

En función de la complejidad de las UoLs a especificar en IMS LD se han considerado tres niveles de implementación. Cada uno de estos niveles se describe de forma incremental sobre los conceptos y las relaciones del nivel anterior:

- *Nivel A*. Describe el modelo conceptual que constituye la base de la especificación IMS LD y comprende la definición de actividades de aprendizaje, activi-

dades de soporte, entornos, recursos, método docente, representaciones, actos, roles. Es decir, en este nivel se describe los componentes pedagógicos de una unidad de aprendizaje.

- *Nivel B.* Añade propiedades y condiciones al modelo proporcionada por el Nivel A. Asimismo, también añade servicios de monitorización y elementos globales, lo que permite al usuario crear estructuras más complejas. Las *propiedades* almacenan información acerca de las personas (preferencias, resultados, información personal...), los roles o sobre el propio diseño de aprendizaje. También se contextualizan las propiedades de forma que si son locales se mantienen únicamente durante la ejecución de una instancia mientras que si son globales se pueden consultar y utilizar en diferentes instancias y sus datos persisten a través de varias sesiones. También define un conjunto de operadores y expresiones de forma que el estado de las propiedades y de las condiciones permitan modificar el estado del WF y además influir en el desarrollo de la UoL. También permite esconder y mostrar elementos, condicionar el flujo de aprendizaje, almacenar datos del usuario y la instancia, bien a nivel local y personal, bien a nivel global y compartido.

- *Nivel C.* Añade notificaciones al Nivel B, las cuales se ejecutan automáticamente como respuesta a eventos que se originan en el proceso de aprendizaje. Por ejemplo, si un estudiante envía un trabajo para ser evaluado, se podría enviar automáticamente un correo electrónico al profesor para informarle de tal suceso.

Los tres niveles de la especificación IMS LD están pensados para la definición de unidades de aprendizaje de diferente grado de complejidad. En este sentido, el nivel B es especialmente interesante en la medida en que permite la creación de unidades de aprendizaje adaptativas: el flujo de aprendizaje se adapta a las características y evolución pedagógica de los estudiantes, de modo que cada uno de ellos ejecuta un conjunto de actividades de aprendizaje que no tienen que ser necesariamente las mismas para todos. Por ello, el nivel B de las especificación IMS LD ha sido el que ha atraído más atención de la comunidad investigadora y el motor de unidades de aprendizaje que presentaremos en este capítulo permitirá la ejecución de este tipo de unidades de aprendizaje.

8.3. Motores de ejecución IMS LD

El desarrollo de un motor IMS LD es un proyecto complejo que comparte muchas similitudes con los motores de ejecución de WFs, dado que el diseño de aprendizaje se entiende como la coordinación de la ejecución de actividades ejecutadas por los usuarios (llamados estudiantes y profesores) en un contexto educativo (para conseguir unos objetivos pedagógicos). Además, la especificación IMS LD en sí es muy compleja si la comparamos con otros lenguajes de modelado educativo: la mayoría de estos lenguajes modelan únicamente el concepto de actividad que debe ser realizada por los estudiantes, mientras que en IMS LD introducen más elementos de control

(representaciones, actos, partes de rol, actividades educativas y estructuras de actividades) que tienen restricciones entre sí a la hora de ser ejecutados (por ejemplo, los actos dentro de una representación se ejecutan en secuencia); las restricciones que se deben seguir a la hora de ejecutar cada uno de estos elementos; y la presencia de condiciones y propiedades que personalizan el itinerario educativo

Debido a esta complejidad se han desarrollado relativamente pocos motores de ejecución IMS LD, y actualmente sigue siendo un problema abierto el diseño e implementación de un motor que, por una parte, facilite la personalización de la unidad de aprendizaje a los estudiantes y, por otra, permita comprobar las propiedades del WF para determinar si, por ejemplo, existen puntos muertos o si todos los estados pueden ser alcanzados. Teniendo esto en cuenta, destacamos los siguientes motores de ejecución:

- *CopperCore* [286] ha sido desarrollado por la OUNL para probar la viabilidad de la especificación IMS LD y es la implementación de referencia de IMS LD. Por ello ha sido usado en varios proyectos europeos como TENCompetence [4] o UNFOLD [5] y ha permitido el desarrollado de tecnologías basadas en IMS LD, dado que ha facilitado a los investigadores del campo el desarrollo de herramientas con las que se ha evaluado la especificación.

  CopperCore es capaz de ejecutar documentos IMS LD especificados en cualquiera de los tres niveles de modelado, e internamente traduce el flujo de aprendizaje IMS LD a máquinas de estados finitos cuyo ejecutor está implementado en un lenguaje de programación de Java. Aunque desde el punto de vista conceptual el uso de máquinas de estados finitos es adecuado para la ejecución de WFs, no permite la evaluación de propiedades de forma sencilla, más aún cuando están representadas en Java. Es importante destacar, además, que CopperCore ha externalizado las características del motor de ejecución a través de servicios web, lo que ha permitido su uso en interfaces gráficas, como SLED, basadas en otros lenguajes de programación distintos de Java.

  Por otra parte, CopperCore presenta una interfaz muy pobre y nada intuitiva, que está destinada a la verificación de que las unidades de aprendizaje se ejecutan siguiendo los requerimientos del profesor que las diseña. Esto significa que esta interfaz no es muy adecuada para su uso por parte de usuarios finales en una unidad de aprendizaje real.

- *GRAIL* [99] ha sido desarrollado por la Universidad Carlos III de Madrid y la implementación se ha realizado en el lenguaje .LRN. Actualmente permite la ejecución de documentos IMS LD en cualquiera de los tres niveles de especificación y presenta una intefaz gráfica intuitiva que permite a los usuarios ir ejecutando las actividades de aprendizaje. Desde el punto de vista técnico su principal característica, y limitación, es la integración entre el motor de ejecución y la interfaz gráfica que utilizan los estudiantes para acceder a los contenidos y a las

[4] http://www.tencompetence.com
[5] http://www.unfold-project.net

actividades. Por otra parte, y debido a las características del lenguaje .LRN, presenta ciertos problemas de rendimiento en la ejecución del nivel B cuando el número de propiedades, condiciones y actividades es suficientemente elevado (para unidades de aprendizaje muy complejas).

- Existen otras herramientas como MOT+ [201] y LAMS [90, 143] que están basadas en otros lenguajes de modelado educativo, pero que permiten la ejecución de documentos IMS LD. Para ello realizan una importación en la que traducen del modelo IMS LD a su modelo de representación interno, de modo que el resultado de la ejecución es el mismo: la herramienta le presenta al usuario las actividades de aprendizaje que debe ir ejecutando en cada momento y los contenidos y herramientas de comunicación que debe utilizar en dicha actividad. Sin embargo, dada la complejidad de IMS LD, el soporte de estas herramientas no es completo y, además, no abarca los tres nivel de especificación.

Estos sistemas presentan tres problemas principales. El primero es que no están basados en un formalismo de representación de WFs, con lo que no es posible comprobar las propiedades de la red para asegurar que no tienen puntos muertos, que se puede acceder al estado final desde cualquier estado de la red, etc. El segundo problema es que la ejecución del flujo de aprendizaje está implementada en un lenguaje de programación (como Java o .LRN), lo que significa que cualquier modificación a la semántica de la especificación IMS LD implicará una re-programación del motor de ejecución. Por último, la tercera limitación es que, salvo en Coppercore, existe una fuerte integración entre el motor de ejecución y la interfaz gráfica que permite la visualización de las actividades y recursos educativos.

8.4. Escenario de aprendizaje en astronomía

La estructura de WF que se describirá en las siguientes secciones de este capítulo es compleja de entender si no se aplica a un ejemplo. Por ello, en este apartado se describirá dicha estructura a través de un ejemplo real de unidad de aprendizaje. Esta situación de aprendizaje se ha planteado en el taller *Comparing Educational Modelling Languages on a case study* organizado en el contexto de la Sexta Conferencia Internacional del IEEE sobre tecnologías de aprendizaje (ICALT'06) con el propósito de comparar diferentes técnicas/lenguajes de modelado educacional utilizadas para solucionar un caso de estudio determinado. La actividad propuesta fue escogida a partir de un escenario real de aprendizaje en astronomía.

El objetivo de la actividad es hacer que los estudiantes adquieran una serie de conocimientos en astronomía, en concreto tienen que clasificar planetas de acuerdo a su distancia en relación al sol (del más cercano al más lejano). Para la realización de esta actividad, se ha utilizado un diseño de aprendizaje colaborativo asistido por ordenador teniendo en cuenta los objetivos del aprendizaje: el profesor quiere que los alumnos trabajen juntos, que adopten un método de trabajo y que realicen negociaciones con sus compañeros.

La estrategia utilizada por el profesor para alcanzar estos objetivos es proponer un juego a los estudiantes. Los estudiantes son agrupados en dos equipos (denominados Equipo A y Equipo B). Cada equipo tiene solamente una parte del conocimiento y el conjunto de los datos necesarios para resolver el problema. La meta consiste en resolver una serie de cuestiones para lo que es imprescindible la colaboración entre ambos equipos. Para ello, determinados recursos y servicios (material didáctico sobre los planetas, servicio de chat y foro) estarán disponibles para ayudar a los estudiantes en la adquisición de nuevo conocimiento y en el intercambio de pistas con los miembros de su equipo y en la negociación. Las reglas del juego son las siguientes:

- Distribución de pistas:
  - El equipo A sabe las propiedades de los planetas (obtenidas de entrevistas con los expertos). A partir de esa información pueden deducir el orden de los planetas, pero no saben sus nombres.
  - El equipo B sabe los nombres de los planetas (obtenidas de entrevistas con los expertos) pero, por otro lado, la información de las propiedades es incompleta.
- Los equipos pueden cooperar utilizando un foro para el intercambio de información. Deben, por lo menos, asociar los nombres de los planetas con su posición en relación al sol. Cada equipo puede usar la mensajería instantánea para realizar una discusión entre sus miembros.
- El profesor tiene acceso al foro y puede participar en las discusiones. También puede añadir nuevas pistas actuando como experto sobre el orden de los planetas.
- El profesor decide cuando terminan los intercambios. Entonces, cada estudiante rellena un cuestionario sobre clasificación de planetas.
- El ganador es el que haga una buena asociación *planeta-posición*. La actividad termina cuando se determina un ganador.

La interacción entre los estudiantes se realiza como en una negociación, donde ambos equipos buscan sacar el máximo beneficio. La situación es un tanto paradójica porque para vencer, un equipo tiene que ayudar al otro y, a la vez, tiene que hacer que el otro pierda.

En la Figura 8.2, se describe un diagrama de flujo de las tareas del profesor y de los alumnos durante la realización de la actividad de astronomía propuesta. De acuerdo a la figura, el profesor se encarga de iniciar el proceso, puede añadir información a las entrevistas con los expertos (Añadir Nueva Información), puede participar en el foro (Foro de Discusión) y se encarga de supervisar la ejecución de las tareas de los alumnos (Monitorización). Estas tareas del profesor se agrupan en una estructura representando una secuencia de actividades (Verificar Proceso). Los dos grupos de alumnos (representados como Equipo A y Equipo B) evalúan los documentos con

la información sobre los planetas obtenidas de las entrevistas con expertos (Evaluar Entrevista) y pueden intercambiar información con el foro (Foro de Discusión) o negociar el intercambio de información con la mensajería instantánea (Negociación entre Miembros). Estas tareas de los alumnos se agrupan en una estructura representando una secuencia de actividades (Resolver Nombre y Orden) y se realizan en iteraciones en paralelo, controladas por el profesor. Cuando el profesor decide terminar la actividad de intercambio de información, asigna un parámetro en la actividad de *Monitorización* dentro del grupo de actividades *Verificar Proceso*, parando el proceso y dando inicio a la actividad *Cuestionario*.

8.5. Flujos de trabajo para la ejecución de unidades de aprendizaje

Como resultado del proceso de diseño, OPENET LD da soporte a un conjunto de WFs para la tarea de presupuestado de muebles [279]. La Figura 8.3 representa a la HLPN que se encargará de controlar la ejecución de los método docentes, representaciones, actos y actividades de la unidad de aprendizaje. Por lo tanto, ya que una unidad de aprendizaje se ejecuta mediante un método docente, esta HLPN constituye el punto de partida de cualquier WF de aprendizaje.

En este trabajo se ha usado una estructura común para unificar el modo en que un método docente, una representación, un acto o una actividad pueden ser ejecutados, parados, suspendidos, reanudados, pausados o finalizados. De esta forma, todos los elementos que constituyen la UoL podrán pasar por la misma estructura de estados. Como se puede observar en la Figura 8.3, esta red se construye a partir de un patrón *secuencia*. La primera parte de esta secuencia está dedicada a la ejecución y control de los distintos estados por los que puede pasar un determinado elemento. Esta parte se implementa a través del patrón *separación-unión* y consta de cuatro ramas en paralelo:

- La primera rama muestra la transición encargada de ejecutar una determinada estructura. En función del tipo de elemento, esta transición será sustituida por una red diferente. Por ejemplo, si se trata de un método docente será sustituida por una red que controle su conjunto de representaciones; si se trata de una representación, por un conjunto de actos a ejecutar; si se trata de un acto, por un conjunto de entidades de ejecución; si se trata de una actividad con estructura, por la red que representa dicha estructura; finalmente, si se trata de una actividad sin estructura, se asociará el método primitivo encargado de resolver la actividad.

- La segunda rama se encarga de modelar el ciclo de suspensión y reanudación del elemento. Esta rama se modela a través de un patrón *repetir-mientras*. Por lo tanto, si se cumplen unas determinadas condiciones entonces el cuerpo del bucle estará activo. En este caso las condiciones de la transición *suspender elemento* se refieren al estado de ejecución del elemento. Si el elemento está en el

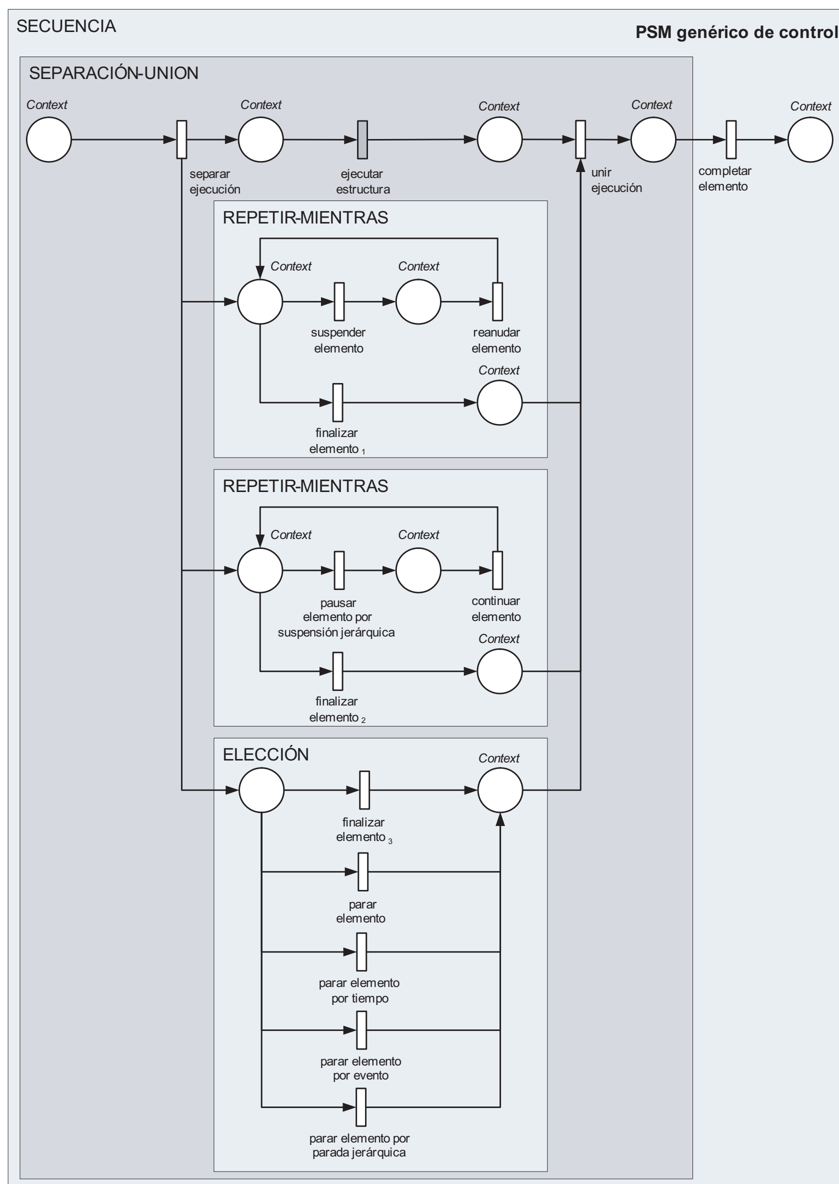


Figura 8.3: Estructura genérica que se encargará del control de la ejecución de los método docentes, representaciones, actos y actividades de las unidades de aprendizaje

estado *ejecutándose*, la transición estará activa y la ejecución del elemento podrá suspenderse. El operador *isRunning* anota la condición de guardia de esta transición y toma como entrada al *contexto de ejecución* y al identificador del elemento. Internamente, la función encargada de ejecutar dicho operador mirará

el estado del elemento en el contexto de ejecución y actuará en consecuencia. El arco de salida de esta transición está anotado con el operador *suspend*. La evaluación de este operador cambiará el estado del elemento en el contexto de ejecución a *suspendido*. El siguiente paso del cuerpo del bucle es la transición *reanudar elemento*. La ejecución de transición se encargará de cambiar el estado de ejecución del elemento de suspendido a *ejecutándose*. El operador *resume* que anota el arco de salida de esta transición permitirá este cambio. Finalmente, cuando el estado del elemento pasa al estado de *finalización*, la transición *finalizar elemento*$_1$ se encarga de finalizar la ejecución del bucle.

- Al igual que la segunda rama, la tercera se encarga de modelar el ciclo de suspensión y reanudación del elemento pero en este caso motivado por la suspensión o reanudación de alguno de los elementos jerárquicamente superiores al elemento modelado. Por ejemplo, la suspensión de un método docente provoca que todas sus representaciones deban suspenderse también. La transición *pausar elemento por suspensión jerárquica* se encarga de esta tarea. Esta transición está anotada por la precondición *isFatherSuspended* que verifica si el padre está suspendido. Cuando el padre reanuda su ejecución, la ejecución de la transición *continuar elemento* se encargará de devolver el estado del elemento a *ejecutándose*.

- La última rama de la estructura se encarga de parar la ejecución del elemento modelado. Existen cinco condiciones por las que un elemento pueda pararse. Por este motivo, esta rama se modela por medio de un patrón *elección*. La primera razón por la que puede pararse un elemento es porque su ejecución haya finalizado. ¿Tiene esto sentido? Desde el punto de vista del modelado sí. Supóngase que se haya completado la transición *ejecutar estructura*. En ese momento, el objetivo sería ejecutar la transición *completar elemento* pero su activación depende de la finalización de cada una de las ramas paralelas. Para finalizar cada una de estas ramas se hará uso de las transiciones *finalizar elemento*$_i$.

  La segunda razón de finalización es por decisión del usuario. La transición *parar elemento* se refiere a esta situación. La ejecución también puede pararse cuando un elemento ha superado un determinado tiempo límite (transición *parar elemento por tiempo*). Otra forma de parar la ejecución se da cuando se verifica un determinado evento. Estos eventos se refieren a los valores que pueden tomar las propiedades definidas en el nivel B del IMS LD. Cuando una propiedad toma un determinado valor, el elemento puede finalizar su ejecución. Finalmente, la última forma de parar la ejecución de un elemento es porque su padre haya finalizado su ejecución.

La segunda parte de la secuencia del método genérico de control se encarga de completar la ejecución del elemento. En las unidades de aprendizaje, algunos elementos generan un cambio en las propiedades y variables almacenadas en el contexto de ejecución. Por esta razón, el arco de salida de la transición *completar elemento* se anota con la llamada al operador *onCompletion*. Por ejemplo, la ejecución de un determinado elemento puede hacer visible una determinada parte de la red de la unidad de aprendizaje a los usuarios. La función que implementa al operador *onCompletion* se

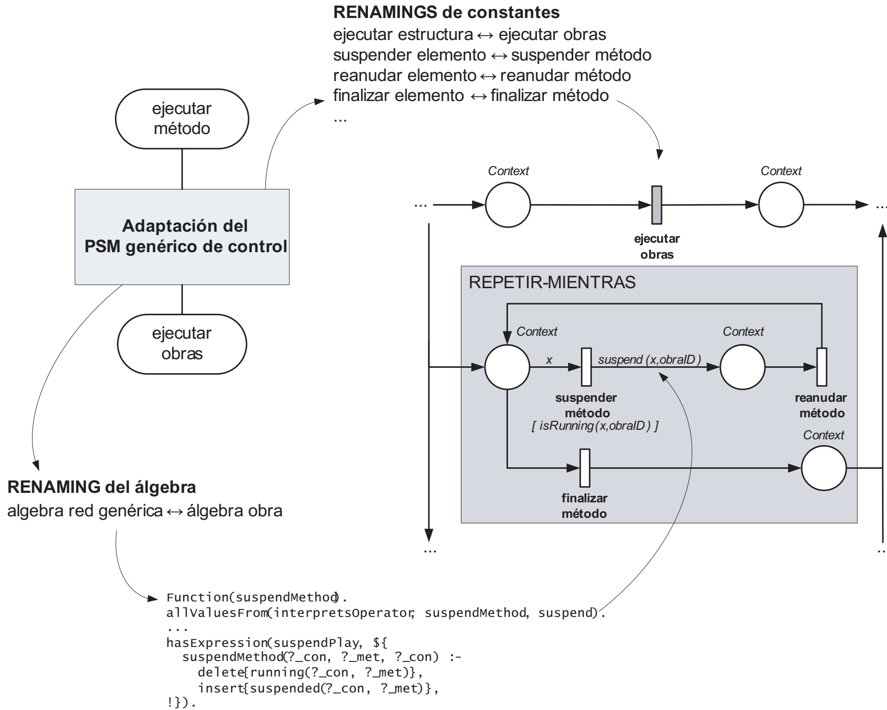


Figura 8.4: Esquema de los elementos que participan en la suspensión de un método docente

encarga además de realizar una nueva evaluación de las condiciones definidas para la ejecución de la unidad de aprendizaje.

A modo de ejemplo, la Figura 8.4 representa la adaptación del método a las características del método docente. Esta adaptación trata *(i)* de adaptar las constantes que difieren entre ambos modelos: las plazas, transiciones y demás identificadores que anotan la red. Por ejemplo, la transición *ejecutar estructura* se sustituye por la transición *ejecutar representaciones*. En este caso, el cambio es importante, ya que afecta a la descomposición en tareas del método compuesto. También establece *(ii)* la nueva álgebra encargada de interpretar la firma de la red. La asociación del álgebra afecta tanto a los colores como a las operaciones que anotan la red. Por un lado, permite definir un determinado contexto de ejecución. Por el otro permite asociar la semántica de interpretación a los operadores. Por ejemplo, la Figura 8.4 muestra esquemáticamente los elementos que intervienen en la suspensión de un método docente. Puede verse en la figura la definición de la función *suspendMethod* del álgebra que se encargará de interpretar las invocaciones del operador *suspend* que anota el arco de salida de la transición *suspender método*. En este caso, la función simplemente elimina el estado previo del método docente e indica que el nuevo estado es *suspendido*. La nueva álgebra también afecta a las constantes que anotan la red. Por ejemplo,

la constante *ID* estará asociada a una función del álgebra que devolverá el método docente.

8.5.1. Estructura de un método docente

Un método docente describe la dinámica de un proceso de aprendizaje y se compone de un determinado número de representaciones. Estas representaciones son independientes y pueden ejecutarse de forma concurrente las unas de las otras. Teniendo en cuenta esta definición, la Figura 8.5 muestra la HLPN que se encarga de controlar la ejecución de cada una de las representaciones que componen el método docente. Esta red sustituirá a la tarea *ejecutar representaciones* incluida en la red de control del método compuesto. La HLPN se basa en el patrón *separación-unión* para lanzar dos ramas concurrentes:

- La primera rama paralela se encargará de ejecutar las n representaciones obligatorias del método docente. Esta ejecución se implementa con un patrón *secuencia* de dos pasos. El primer paso se implementa con un patrón *separación-unión* donde cada una de las n ramas paralelas identifica una representación a ejecutar. El segundo paso de la secuencia se produce cuando todas las ramas han finalizado su ejecución. En ese momento, la transición *completar representaciones* se encargará de indicar en el contexto de ejecución que todas las representaciones obligatorias del método docente han finalizado. Este evento provocará la parada del método docente en la red de control previamente descrita y de paso la detención de las ramas no obligatorias.

- La segunda rama paralela controla la ejecución de las $m - n$ representaciones no obligatorias del método docente. Esta ejecución se implementa con el patrón *separación-unión* donde cada una de las $m-n$ ramas identifican una representación a ejecutar. En este caso, la no obligatoriedad se refiere a que su finalización no es obligatoria más que al hecho de que sean opcionales. Por ello, cuando las representaciones obligatorias finalizan su ejecución, las no obligatorias también finalizan.

Como se puede apreciar en la parte inferior de la figura, el método se descompone en un conjunto de tareas. Cada una de estas tareas identifica a una de las transiciones *ejecutar representación*$_i$. Por lo tanto, durante la ejecución del método cada una de estas transiciones será sustituida por el correspondiente método que resuelve dicha transición. En este caso, el método en cuestión se corresponderá con la estructura de control común y adaptada a la representación a realizar. La parte inferior de esta figura también muestra como la adaptación del método de control se descompone a su vez en una tarea denominada *ejecutar actos de la representación*$_i$. Al igual que un método docente se estructura en base a un conjunto de representaciones, una representación se estructura en función de sus actos. Por lo tanto, la transición *ejecutar actos* (adaptación de la transición *ejecutar estructura* del método común)

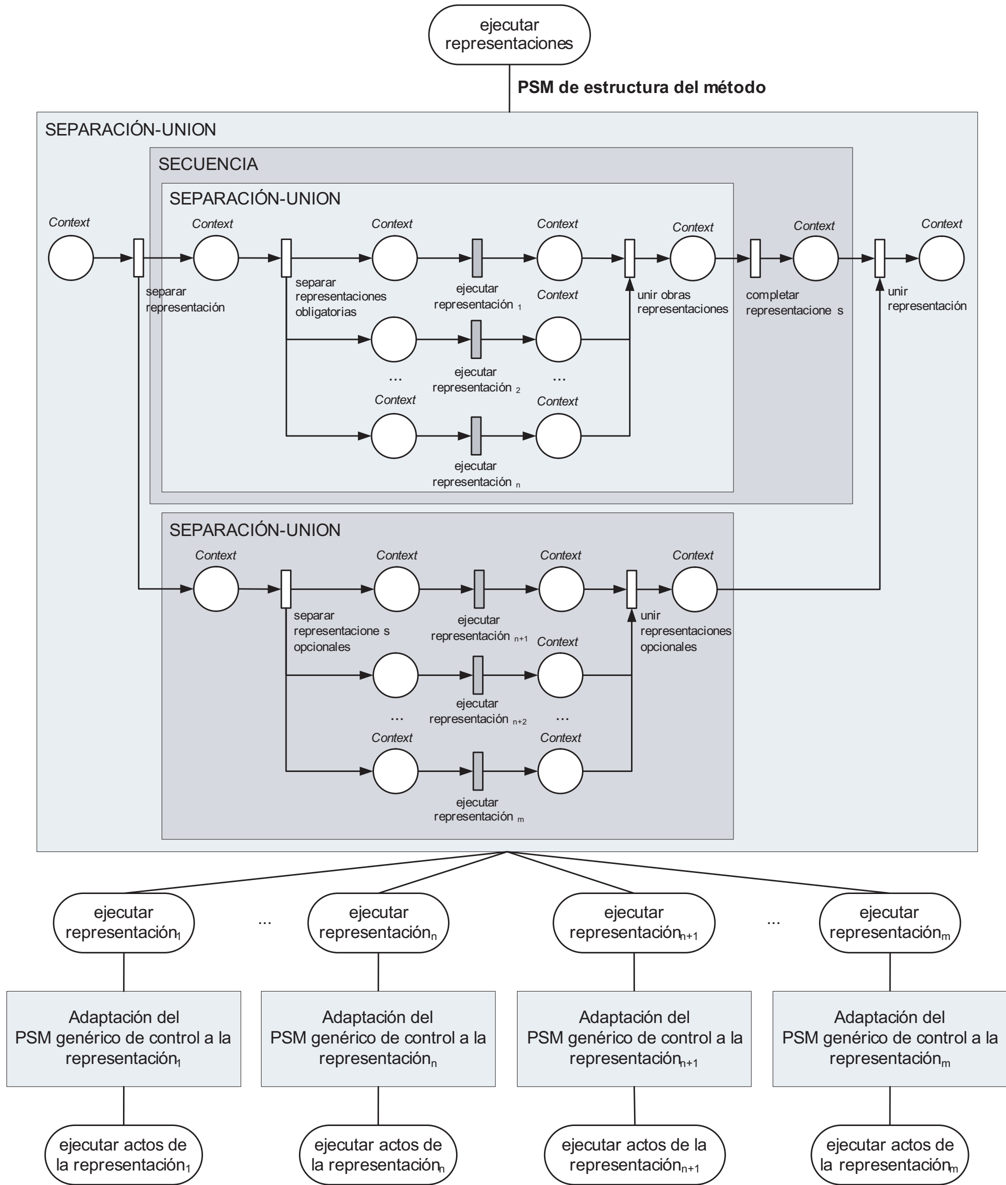


Figura 8.5: Flujo de trabajo que controla la ejecución de un conjunto de representaciones

representará la ejecución de dichos actos y el método compuesto que resuelva la representación identificará dicha transición como una tarea.

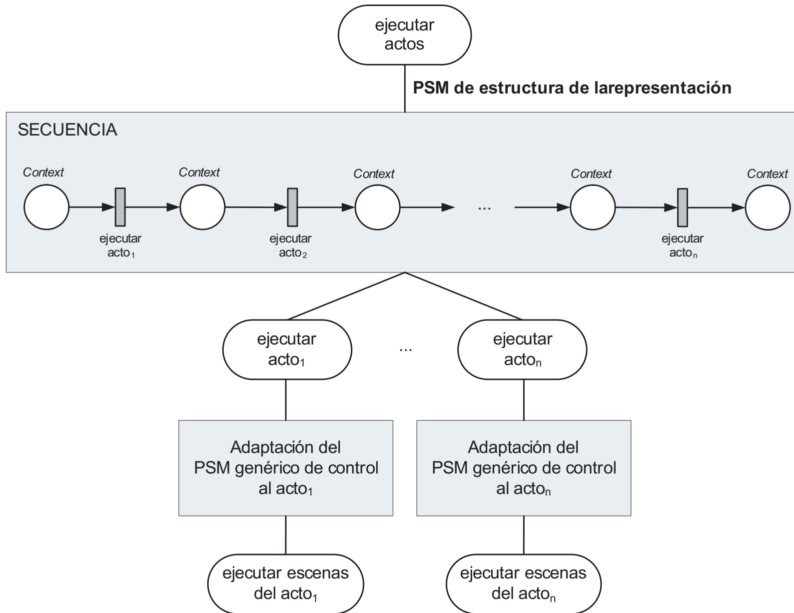


Figura 8.6: Flujo de trabajo que controla la ejecución de una secuencia de actos

8.5.2. Estructura de una representación

Las representaciones modelan el núcleo del diseño de aprendizaje y se describen utilizando la metáfora del teatro. Por ello, una representación consiste en la ejecución ordenada de un conjunto de actos. Por lo tanto, un acto no podrá comenzar hasta que no se complete el acto anterior. El método compuesto que se refiere a este concepto está representado en la Figura 8.6. Este método adapta el patrón *secuencia* donde cada una de las transiciones de control representa un acto a ejecutar.

Como se puede apreciar en la parte inferior de la figura, el método se descompone en un conjunto de tareas. Cada una de estas tareas identifica a una de las transiciones *ejecutar* $acto_i$. Por lo tanto, durante la ejecución del método cada una de estas transiciones será sustituida por un método con una estructura de control adaptada al acto a realizar. La parte inferior de esta figura también muestra como estos métodos se descomponen a su vez en las tareas denominadas *ejecutar escenas del* $acto_i$.

Un acto se estructura en función de unas escenas. Una escena, también denominada *entidad de ejecución* en la especificación IMS LD, es un concepto abstracto que representa una actividad o una unidad de aprendizaje. Por lo tanto, la ejecución de una escena puede dar lugar a la ejecución de un nuevo método docente.

8.5.3. Estructura de un acto

La metáfora del teatro define un acto como un conjunto de *entidades de ejecución/escenas* a ejecutar en paralelo. La estructura de un acto es, por lo tanto, similar a la del método docente y por ello la HLPN que modela esta estructura tiene un fuerte parecido con respecto al método compuesto representado en la Figura 8.5. Sin embargo, difieren en un punto: los actos conocen los actores que van a participar en cada una de sus escenas.

La Figura 8.7 representa el método compuesto que controla la ejecución de un conjunto de escenas. Este método se construye a partir de un patrón *separación-unión* de dos ramas donde la primera rama se encarga de la ejecución de las escenas obligatorias mientras que la rama inferior controla las escenas opcionales. La diferencia respecto a la estructura que controla los métodos docentes son las transiciones *inyectar usuarios de la* $escena_i$ y *extraer usuarios de la* $escena_i$ de las redes. En realidad, estas transiciones representan tareas que forman parte de la descomposición del método y estas tareas se resuelven a través de métodos primitivos. Sin embargo, para no sobrecargar las figuras los método primitivos se representan como simples transiciones. La ejecución de la transición *inyectar usuarios de la* $escena_i$ permite generar una marca por cada uno de los actores que participan en la escena. Este comportamiento se debe a que el arco de salida de la transición *inyectar usuarios de la* $escena_i$ está anotado por un multiconjunto de constantes donde cada una de esas constantes representa a un contexto de ejecución para un determinado actor. De esta forma, cada uno de los actores podrá ejecutar la escena. Otra de las particularidades de la red son las transiciones *extraer usuarios de la* $escena_i$. Su propósito es justo el contrario del de la tarea anterior y consiste en finalizar la ejecución de la escena cuando todos los actores la hayan finalizado. En este caso se consigue anotando el arco de entrada de la transición con un conjunto que contiene los contextos de ejecución de los actores. El resto de la HLPN es idéntico al descrito para la estructuración del método docente.

La parte inferior de la figura muestra la descomposición del método y como cada una de las tareas *ejecutar* $escena_i$ se resuelve adaptando el método genérico a la escena i. Es importante mencionar que la figura no muestra la descomposición en tareas de los métodos de control de las escenas. Ello se debe a que una escena puede resolverse mediante una actividad o mediante otra unidad de aprendizaje. Por lo tanto, dependiendo de si es una u otra, la descomposición del método genérico adaptado puede ser diferente.

8.5.4. Estructura de una escena

Una escena puede ser una actividad u otra unidad de aprendizaje distinta a la que se está ejecutando en ese instante. Cada una de estas escenas, independientemente de su tipo, se resuelven mediante el método genérico adaptado a sus características. Sin embargo, la descomposición de este método diferirá en función de las características de la escena. Cuando la escena es una actividad, caben tres posibles comportamientos:

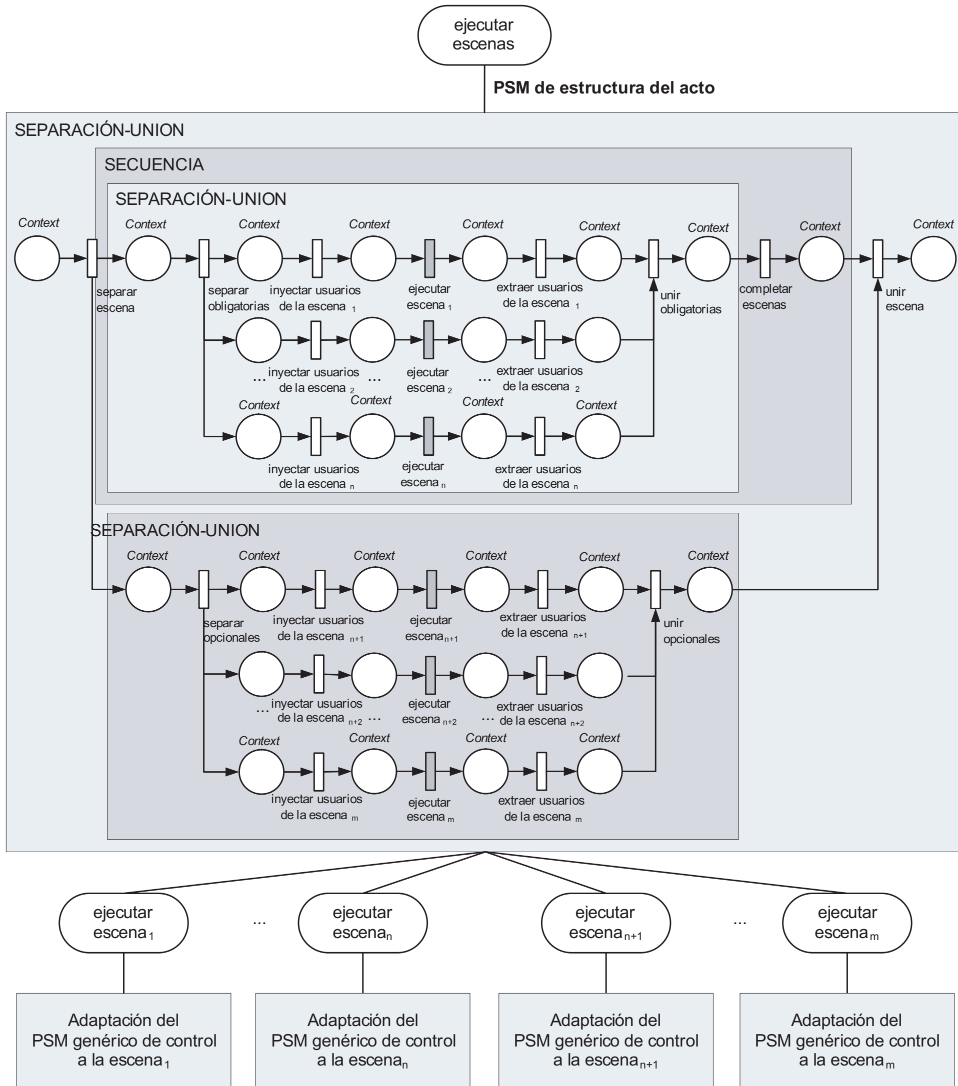


Figura 8.7: Flujo de trabajo que controla la ejecución de un conjunto de entidades de ejecución

- *Actividades simples.* Es el tipo más sencillo de actividad. Representa una actividad educacional que no se descompone. Por lo tanto, constituyen las hojas del árbol de composición de los métodos del diseño de aprendizaje. Al igual que los demás elementos, también se modela a través de la adaptación del método genérico. Sin embargo, en este caso no sustituye la transición *ejecutar estructura* con una estructura más compleja. Como está representado en la Figura 8.8, esta transición se resuelve por un método primitivo. Por lo tanto, y simplificando la

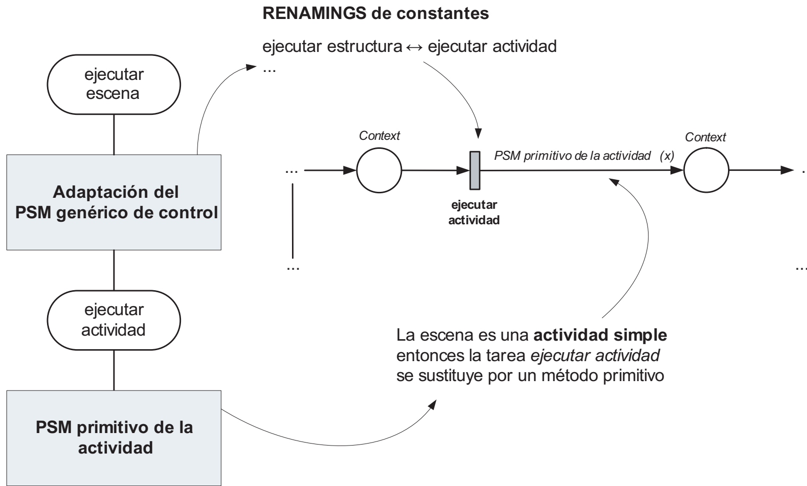


Figura 8.8: Flujo de trabajo que controla la ejecución de una actividad simple

composición de la HLPN, bastaría con anotar el arco de salida de la transición con la invocación de ese método.

- *Actividades con una estructura en secuencia.* Es un tipo de actividad con una estructura secuencial donde cada uno de los elementos de la secuencia será una escena. Por lo tanto, este tipo de actividades representa un nodo del árbol de descomposición de la tarea. La Figura 8.9 muestra la estructura de este tipo de actividades donde se puede apreciar que se impide el inicio de la $escena_i$ mientras no haya finalizado la $escena_{i-1}$, $i = 1 \ldots n$. Cada una de las escenas se resolverá a través de la adaptación del método genérico a la escena a resolver. La descomposición de este método dependerá del tipo de escena.

- *Actividades con una estructura de selección.* Algunas actividades del IMS LD se pueden estructurar como una selección de un conjunto de escenas. En este caso, la estructura dispone de un conjunto de escenas que se pueden seleccionar y la actividad finaliza cuando un determinado número de ellas ha finalizado. La Figura 8.10 muestra la HLPN que controla este tipo de actividad. Esta actividad se implementa mediante un patrón *repetir-mientras* donde el cuerpo del bucle contiene al conjunto de escenas que se pueden seleccionar. Cada vez que se selecciona una escena mediante una transición *seleccionar* $escena_i$, se indica al contexto que la escena ha sido seleccionada y se reduce el número de escenas que faltan por seleccionar en una unidad. Cuando dicho número sea igual a cero, la transición *finalizar selección* estará activa y su ejecución finalizará el bucle.

Cuando la escena es otra unidad de aprendizaje, la transición *ejecutar unidad de aprendizaje* (adaptación de la transición *ejecutar estructura* del método común) será sustituida por la estructura del método que contiene el diseño de aprendizaje de la

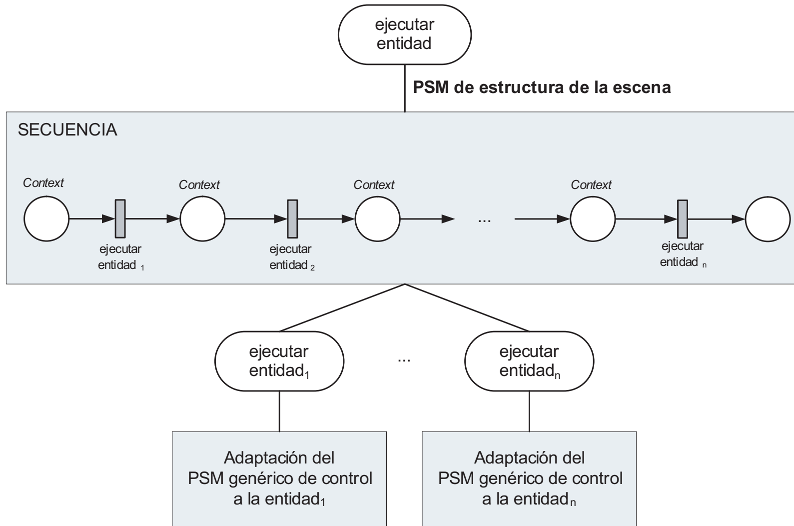


Figura 8.9: Flujo de trabajo que controla la ejecución de una secuencia de entidades de ejecución

unidad. Se puede decir que un (sub)árbol colgará a partir de este nodo del árbol, ya que del método colgarán las representaciones, de las representaciones los actos y de los actos las escenas, y así sucesivamente hasta que todas las ramas finalicen con actividades simples. Comentar también que no existe límite a esta composición. Por ejemplo, esta segunda unidad de aprendizaje puede a su vez contener escenas que son otras unidades de aprendizaje.

8.6. Arquitectura

La arquitectura software del ejecutor de IMS LD nivel B sigue el paradigma de las arquitecturas orientadas a servicios para facilitar el mantenimiento y desarrollo independiente tanto del motor de ejecución IMS LD nivel B como de la interfaz gráfica que utilizarán los usuarios para acceder a la información de la unidad de aprendizaje [280]. La Figura 8.11 muestra el esquema de componentes de la arquitectura:

- Una capa de clientes correspondiente a la interfaz gráfica que los usuarios usan para visualizar las actividades a ejecutar en cada momento. Con esta interfaz gráfica los usuarios pueden *(i)* acceder a los recursos y servicios de las actividades que deben realizar, *(ii)* solicitar acciones sobre el motor de ejecución para indicar la finalización de una determinada actividad, y *(iii)* visualizar las actividades que ya han finalizado.
- La segunda capa es la responsable de resolver las interconexiones entre los elementos de la arquitectura, particularmente entre los servicios web y los clientes

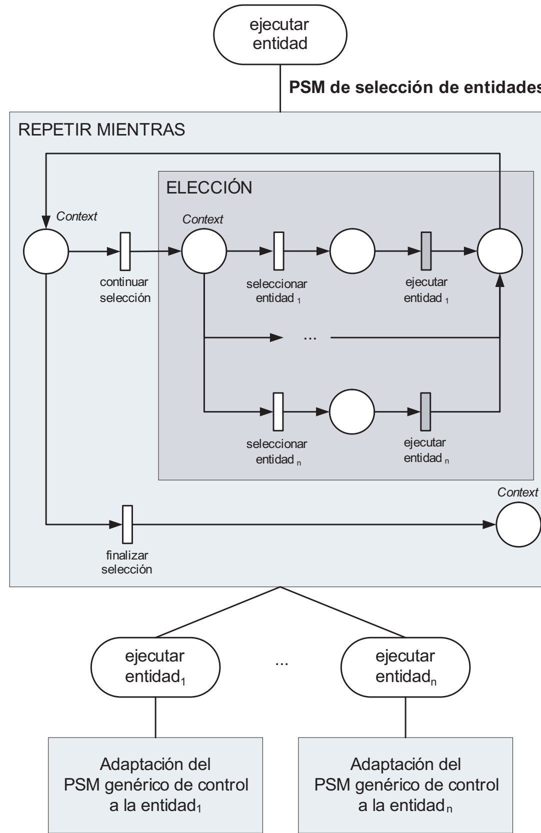


Figura 8.10: Flujo de trabajo que controla la ejecución de una selección de entidades de ejecución

[285]. Para ello se ha usado OpenESB como servidor de aplicaciones e integrador de servicios debido fundamentalmente a que *(i)* ofrece uno de los mejores rendimientos de entre los servidores de aplicaciones open source a la hora de resolver la invocación de los servicios; *(ii)* integra módulos (service engine) de procesamiento como, por ejemplo, la composición de servicios a través del lenguaje BPEL, y componentes para el tratamiento de protocolos de comunicación (binding components) que pueden utilizar los clientes para acceder a los servicios web; y *(iii)* está integrado con la plataforma J2EE y el entorno de desarrollo Netbeans, facilitando enormemente el depliegue de servicios web implementados en Java. Por otra parte, OpenESB define un bus (Normalized Message Router, NMR, en inglés) que es usado por los diferentes módulos y componentes para comunicarse entre sí a través de los patrones de intercambio de mensajes especificados en WSDL 2.0. Esto mejora la interoperabilidad y facilita la integración de diferentes tecnologías y protocolos de diferentes vendedores.

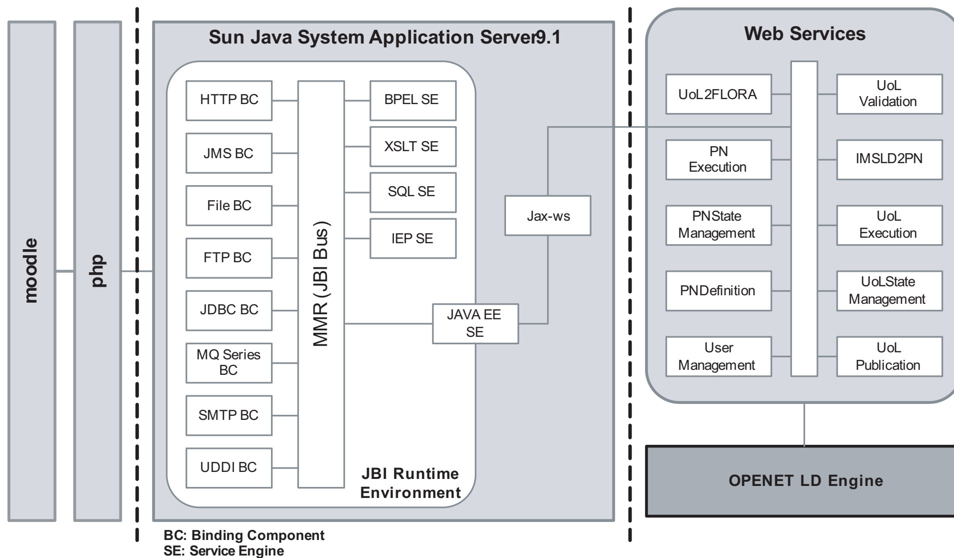


Figura 8.11: Arquitectura orientada a servicios que externaliza las capacidades del motor de IMS LD nivel B

- La tercera capa contiene al conjunto de servicios web que implementan la lógica de negocio del motor de ejecución de IMS LD nivel B. La arquitectura software del motor favorece la externalización de sus funcionalidades, en la medida en que están declaradas a través de los métodos de fachadas de Java: es decir, los servicios web invocan directamente a los métodos de la interfaz de Java del motor IMS LD. Así, tal y como se puede observar en la Figura 8.11, la arquitectura orientada a servicios ofrece 21 servicios web que se pueden clasificar en los siguientes tipos:

    - *Gestión de usuarios* (UserManagement). Permiten la creación de los usuarios de la unidad de aprendizaje y su asignación un determinado rol o roles correspondientes.
    - *Validación semántica* (UoLValidation). Permiten realizar la validación de las unidades de aprendizaje para comprobar que cumplen con la semántica de la especificación IMS LD que está recogida en los axiomas de una de las ontologías manejadas por el motor. Para realizar esta validación se ha creado una colección de servicios web para comprobar que la UoL cumple con la axiomática del modelo.
    - *Gestión de las sesiones y ejecución* (UoL2FLORA, IMSLD2PN y UoLExecution). Permite a los clientes la importación de la unidad de aprendizaje a partir del fichero XML, la transformación a su representación en HLPNs, y la creación de una instancia de ejecución de la unidad de aprendizaje.

Estos servicios usan internamente los servicios web para la creación (PN-Definition) y la ejecución (PNExecution) de las HLPNs.

- *Control de ejecución de OPENET LD* (UoLStateManagement). Permite a los clientes gestionar el estado de los componentes del IMS LD a través del estado de la HLPN asociada a dichos componentes. Más específicamente, con estos servicios es posible *(i)* obtener el estado del flujo de aprendizaje; y *(ii)* invocar la ejecución de las acciones (suspender, reanudar, suspender definitivamente y finalizar) con las que se modifica el estado de los componentes del flujo. Estos servicios utilizan internamente los servicios web (PNStateManagement) con los que se maneja el estado de las redes de Petri que describen la semántica de ejecución de la unidad de aprendizaje IMS LD.
- *Acceso a recursos* (UoLPublication). Permiten consultar los recursos y servicios asociados a las actividades identificadas en la UoL actualmente en ejecución.

La Figura 8.12 representa la arquitectura del motor que da soporte a la ejecución de unidades de aprendizaje nivel B. El objetivo de este motor es permitir tanto la gestión como la ejecución de unidad de aprendizaje, es decir, la carga de unidades empaquetadas siguiendo el estándar IMS LD y la ejecución de las redes de Petri que modelan su flujo de aprendizaje. Puede verse cómo la arquitectura del motor extiende horizontalmente aquella descrita para el motor OPENET4WF. La nueva pila añade la semántica asociada a las ontologías IMS LD nivel A, nivel B y de competencia de los estudiantes. De forma más detallada, los principales elementos de la pila son los siguientes:

- *Esquema*. Almacena el conjunto de clases, relaciones, propiedades y axiomas que describen las unidades de aprendizaje IMS LD, representando en F-Logic la ontología IMS LD nivel B.
- *Datos*. Este módulo almacena las instancias de las unidades que están basadas en la especificación IMS LD. Por ejemplo, almacena las condiciones, las propiedades que definen las características del perfil de estudiante, los componentes del flujo de aprendizaje IMS LD, etc.
- *Reglas*. El conjunto de reglas definidas en la capa IMS LD permite *(i)* crear, modificar y borrar unidades de aprendizaje; *(ii)* evaluar las condiciones y expresiones que caracterizan el nivel B. Por ejemplo, la siguiente regla permite evaluar, tal y como está definido en el IMS LD nivel B, la expresión que realiza la suma de operandos:

```
%evaluateExpression(?_exp, ?_usr, ?_res) :-
  ?_exp:sum_op,
  if   (debug(true))
  then (format("Evaluation info: ~q SUM expression \n",
        [?_exp])@_prolog(format)),
```

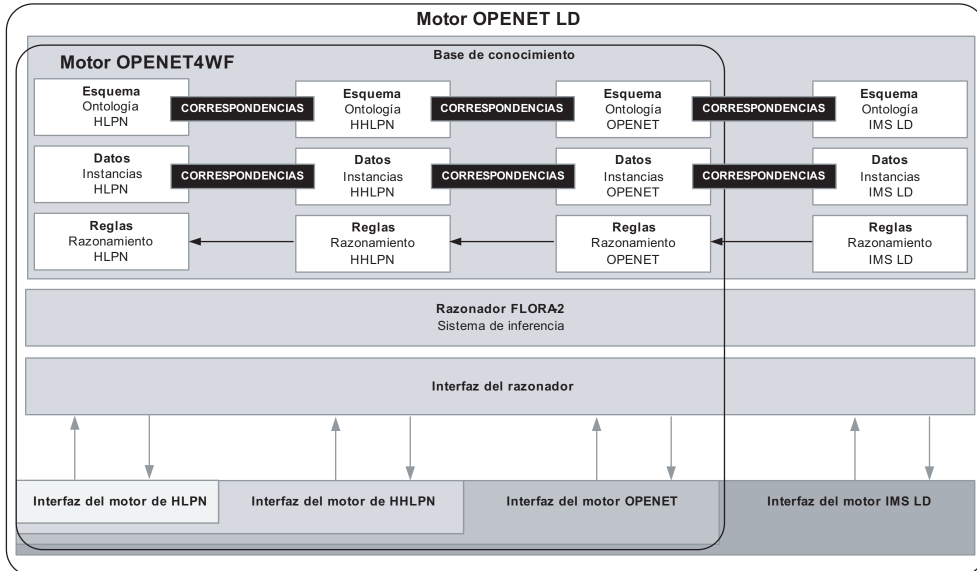


Figura 8.12: Arquitectura del motor de unidades de aprendizaje IMS LD nivel B. Este motor está construido sobre el motor del marco de conocimiento y añade una capa para el manejo del IMS LD nivel B.

```
?_exp[operand1 -> ?_op1, operand2 -> ?_op2],
%evaluateOperand(?_op1, ?_usr, ?_re1),
%evaluateOperand(?_op2, ?_usr, ?_re2),
if (isNumber(?_re1, ?_nu1, true), isNumber(?_re2, ?_nu2, true))
then (?_res is ?_nu1 + ?_nu2)
else (?_res = null,
      if   (debug(true))
      then (format("Evaluation warning: ~q SUM expression \n",
           [?_exp])@_prolog(format))).
```

Además, esta capa proporciona las reglas para *(iii)* representar las UoLs como WFs y los adaptadores entre los distintos componentes, es decir, la asociación entre métodos docentes, representaciones, actos y actividades y su correspondiente HLPN detallada en el modelo de control del marco conceptual.

- *Interfaz del motor de IMS LD*. Esta fachada permite la carga y ejecución de unidades de aprendizaje IMS LD desde un cliente. A través de esta fachada es posible cargar la descripción de la unidad de aprendizaje en formato XML-Schema IMS LD y trasladar dicha unidad al modelo interno de Java, y posteriormente a través de la interfaz del razonador, al lenguaje F-Logic. Esta interfaz se encarga de crear los objetos del modelo completo del flujo de IMS LD nivel B, así como las condiciones y propiedades del modelo. En esta fachada también se realiza el paso del modelo IMS LD al modelo de WF a través de la invocación de un conjunto de reglas que posibilitan crear los componentes del marco conceptual

en función de las definiciones del IMS LD. Finalmente, cabe mencionar que las unidades de aprendizaje se ejecutarán como cualquier otro método a través del gestor de ejecución de OPENET4WF.

8.7. Conclusiones

En este capítulo hemos presentado un motor de ejecución de UoLs especificadas en el estándar IMS LD, denominado OPENET LD, que se ha construido sobre el marco conceptual descrito en esta tesis doctoral [280]. Con este motor las UoLs se ejecutan como WFs facilitando así la coordinación entre las tareas y los participantes que las ejecutan.

El motor da respuesta a las dos principales carencias de las implementaciones de los motores de UoLs existentes en la actualidad. Por un lado, modela formalmente la semántica operacional de cada uno de los elementos de una UoL, es decir, de los métodos docentes, las representaciones, los actos, las escenas o las actividades. Cada elemento se representa a través de una HLPN que captura la semántica descrita en el nivel A del estándar IMS LD. Su descripción mediante patrones de WFs resuelve la forma de tratar la concurrencia y la sincronización de las distintas partes de una UoL y también permite establecer dónde se inyectan los profesores/alumnos durante la ejecución y cómo se anotan las redes con los usuarios para establecer las condiciones de finalización de los actos. Por otro lado, el modelo descrito en este capítulo permite ejecutar UoLs especificadas en el nivel B del IMS LD. Este paso se consigue traduciendo la especificación IMS LD a una ontología [12] y anotando las redes con la firma del nivel B. Estos operadores permiten establecer las condiciones iniciales de ejecución de la red, mostrar y ocultar actividades, cambiar propiedades del contexto de ejecución y evaluar nuevamente a las condiciones una vez que el método docente, representación, acto u escena hayan finalizado. La unión entre la ontología y las anotaciones se consigue a través del álgebra definida para estas redes.

El modelo descrito se caracteriza por partir en dos redes la semántica operacional de los métodos docentes, representaciones y actos. La primera red es común y controla la ejecución del elemento, es decir, controla su parada, suspensión o reanudación. La segunda es específica de cada tipo de elemento y coordina la ejecución de sus (sub)elementos. En el caso de un método docente esta red facilita la ejecución en paralelo de las representaciones; en el de una representación, la ejecución secuencial de un conjunto de actos; y finalmente en el de un acto, la ejecución en paralelo de actividades con y sin estructura o nuevas UoLs.

Como cualquier otro WF, las distintas redes que describen una UoL se combinan en una red jerárquica, en la que se establecen *(i)* los nodos que han de ser sustituidos por otra red, por ejemplo la asociación entre la red de control y la estructura del elemento, y *(ii)* los nodos compartidos entre las distintas redes. Por ejemplo, los nodos que indican el estado de un método docente, de una representación o de un acto son compartidos entre redes diferentes a través del mecanismo de fusión de nodos, permitiendo jerarquizar el control de la UoL. Por ejemplo, la red que controla una

representación observa el estado del método; la red que controla un acto observa el estado de la representación; y finalmente, la red de control de las actividades observa a los actos. Así, cuando un elemento cambia de estado, sus (sub)elementos reaccionan en cadena.

Finalmente, cabe mencionar que a través de esta aplicación se muestra cómo OPENET LD se extiende horizontalmente a OPENET4WF añadiendo una nueva capa que le permite gestionar la ontología IMS LD de nivel A y B. Al haber sido construido sobre un razonador lógico, el añadir una nueva capa a la arquitectura del motor es un proceso directo, ya que basta con introducir las reglas que definan las correspondencias entre el dominio de las UoLs y el de los WFs. En este caso, estas correspondencias establecen las redes que se ajustan a cada elemento, el modelo de recursos a aplicar así como el álgebra de nivel B.

En la actualidad OPENET4WF también este motor está siendo usado como base para la ejecución de UoL dentro del contexto de los mundos virtuales. Los primeros resultados a este respecto pueden encontrarse en [106, 105].

9

Motor de ejecución de servicios OWL-S

En este capítulo se muestra una tercera aplicación [277] del marco conceptual de flujo de trabajo (WF) presentado en esta tesis doctoral. A diferencia de las aplicaciones descritas en los capítulos 7 y 8, donde se refleja un modelo de WF que soluciona una determinada funcionalidad, aquí describimos la utilización de nuestro motor para dar soporte a la ejecución de *servicios web semánticos* expresados en OWL-S [174]. Es pertinente precisar que se denominan semánticos, ya que tanto su especificación en el lenguaje de representación de ontologías OWL [84] como su estructuración permiten a los agentes razonar o inferir conocimiento acerca del servicio. Teniendo en cuenta que la especificación OWL-S ya ha sido descrita en los capítulos 1 y 2, este Capítulo se centra en la adaptación de OPENET4WF para el modelado y ejecución de servicios web semánticos.

El desarrollo del capítulo tiene dos partes claramente diferenciadas. El núcleo del capítulo es la representación a través de redes de Petri de los servicios OWL-S dentro de nuestro metamodelo. Ello es una importante aportación, ya que nos permite *(i) formalizar* la semántica operacional de las construcciones de control definidas en OWL-S (una de las principales carencias de esta especificación) y *(ii)* validar y verificar las propiedades estructurales de los servicios OWL-S. Sin embargo, en este capítulo también se cubre la integración dentro del metamodelo de WFs de cada uno de los tres modelos que definen un servicio OWL-S (ya descritos en el Capítulo 1), es decir, el perfil del servicio, el modelo de servicio, y el modelo de conexión.

9.1. Estado del arte de la formalización de OWL-S mediante redes de Petri

El modelado de OWL-S a través de redes de Petri ha suscitado mucho interés por parte de la comunidad científica en los últimos años. El primer intento de formalización está recogido en [191]. Esta propuesta modela DAML−S (una versión anterior a la especificación actual de OWL-S y representa los servicios mediante redes de Petri de bajo nivel. Por lo tanto, no toma en cuenta los datos y ni las condiciones que restringen el flujo de los servicios. Esta aproximación únicamente representa las estructuras de control que permiten definir la coreografía de servicios DAML−S. Aunque gran parte

de estas estructuras de control son aplicables a la especificación actual de OWL-S, carece de algunas construcciones como por ejemplo el patrón *sin-orden*. Además, considera que el control está compuesto únicamente por procesos atómicos y, por lo tanto, no define los mecanismos de composición entre las distintas estructuras.

En [89], los servicios OWL-S también se modelan con redes de bajo nivel, y por ello su modelo tampoco tiene en cuenta las condiciones de los procesos, ya que las marcas no tienen asociado un color. Sin embargo, su aproximación separa explícitamente el flujo de control del flujo de datos. Las transiciones que identifican al mismo proceso, tanto en el flujo de datos como en el flujo de control, son el punto de unión entre ambos modelos. La unión consiste en la fusión de ambas transiciones de forma que se unen en una única red. El inconveniente de esta aproximación es que complica la estructura de las redes, ya que las plazas deben ser replicadas cuando dos procesos acceden a los mismos parámetros. Esta característica también dificulta la verificación de propiedades y no evita las *lecturas sucias*, ya que las plazas que representan un mismo parámetro no están sincronizadas. En cambio, al contrario de [191], esta aproximación sí permite la composición de procesos a través de la sustitución del proceso compuesto por una subred.

En [34], la representación se basa en *redes Consume-Produce-Read* (CPR). Estas redes están equipadas con dos conjuntos disjuntos de plazas, denominadas plazas de control (sus marcas se pueden producir y consumir) y plazas de datos (sus marcas se pueden producir y leer). Aunque esta aproximación también distingue entre el flujo de control y el flujo de datos, es diferente a la presentada en [89]. Su solución representa un parámetro de OWL-S mediante una única plaza de datos dentro de un canal de comunicaciones compartido de forma que la plaza no necesita ser replicada ni sincronizada. Sin embargo, esta solución hace que una misma plaza esté relacionada con muchas transiciones (una relación por cada entrada o salida donde se utilice el parámetro) de forma que sea difícil conseguir que la red esté *bien-estructurada* y *libre de conflictos* [264]. Las redes CPR tampoco representan el color de los parámetros de forma que no se pueden incorporar al modelo las clases de OWL-S que tipan los parámetros ni las condiciones sobre esos parámetros. Al igual que en [89], la composición se realiza a través de la sustitución de transiciones.

En [308] se define un conjunto de agentes software para facilitar la automatización de la publicación, descubrimiento y ejecución de servicios OWL-S. Uno de estos agentes verifica la publicación a través de *Web Service Petri nets* (WSPNs), que son un tipo de red de Petri de bajo nivel. En esta aproximación, solamente se tiene en cuenta al flujo de control del servicio y, al contrario de las propuestas anteriores, las plazas representan la ejecución de los procesos. Por lo tanto, cuando una marca está en una plaza, un proceso está ejecutándose en un determinado estado. Sin embargo, aunque esta aproximación pueda resultar de cierta utilidad a la hora de verificar las propiedades de la red, no debería usarse para su ejecución, ya que las transiciones no disponen de la información necesaria para averiguar cuando un proceso está ejecutándose o ya ha finalizado.

La aproximación seguida en [64] modela a los procesos OWL-S mediante *redes de Petri coloreadas* (CPN). En esta aproximación, las marcas representan parámetros

o condiciones de OWL-S y no se definen marcas de control. Por ello, el valor de las marcas que identifican a los parámetros y condiciones determinará el flujo de control. En [178] se presenta una aproximación que utiliza ontologías de redes de Petri para representar las entradas y salidas de los parámetros del proceso así como sus condiciones. Sin embargo, no incluyen ninguna referencia acerca de estas ontologías de forma que no se puede evaluar ni la estructura ni el álgebra de estas redes. Desde la perspectiva de las redes de Petri, su modelo almacena los datos del proceso en las plazas y añade las precondiciones a las transiciones. Estas dos soluciones orientadas a datos mantienen las desventajas de [89], llegando incluso a complicar más la replicación de datos y la sincronización de los parámetros al no estar explícito el flujo de control. Los autores tampoco describen cómo las condiciones pueden evaluarse ni tampoco cómo las marcas (que representan los parámetros) se utilizan para evaluar dichas condiciones. En el caso de [64], esta evaluación esta asociación es más compleja, ya que las precondiciones también son marcas.

En [14] se presenta una solución para el modelado de *Workflow Web Services* (W2S). Esta solución representa servicios W2S mediante *redes de Petri con objetos* (PNO) y deriva de esta definición un servicio OWL-S a través de un conjunto de reglas. En este caso, utilizan OWL-S en lugar de otro lenguaje de servicios porque se amolda mejor a la ejecución de servicios W2S. Por ello, únicamente describen cómo modelar en OWL-S las estructuras de control de W2S.

En [183] se modela la coreografía de los servicios OWL-S, tomando como punto de partida la propuesta de [191], pero a través de *Reference Nets* (RNs) que son una extensión del formalismo de las CPNs que representan el flujo de control de los servicios. En estas redes todas las plazas están en un canal de comunicación, de forma que no es necesario replicar ni sincronizar los parámetros del proceso. Aunque resuelve los problemas de [191], su modelado se queda en el nivel de servicios DAML−S. Además, no dan una solución ni analizan en profundidad la forma de representar los datos ni las precondiciones de los procesos.

En [81] se utiliza una metodología para definir las correspondencias entre un servicio OWL-S y un grafo CPN. Utilizan la ontología OWL-S para representar el flujo de trabajo y definen un algoritmo para trasladar el servicio OWL-S a CPN a través de PNML. Esta aproximación no incluye todas las construcciones de control de OWL-S y algunas de sus modelos no cumplen con las restricciones de la especificación OWL-S. Al igual que [64], representan los parámetros de entrada y salida, las precondiciones y los efectos mediante las marcas de la red, sin embargo los autores no precisan cómo se integran en la red ni cómo se evalúa ésta.

Resumiendo, ninguna de las aproximaciones considera el modelado completo de la coreografía definida por la especificación OWL-S. Estas aproximaciones sólo representan las construcciones de control que estructuran los procesos pero no representan la estructura interna de dichos procesos. Por ello, no permiten representar cómo se invoca un servicio, los efectos de la ejecución del servicio en el entorno, ni las transformaciones en la salida. Además, estas propuestas no describen en detalle cómo los parámetros de entrada, locales o de salida del proceso, ni sus precondiciones, se pueden trasladar al álgebra y a la anotación de la red. En el caso de las propuestas basadas

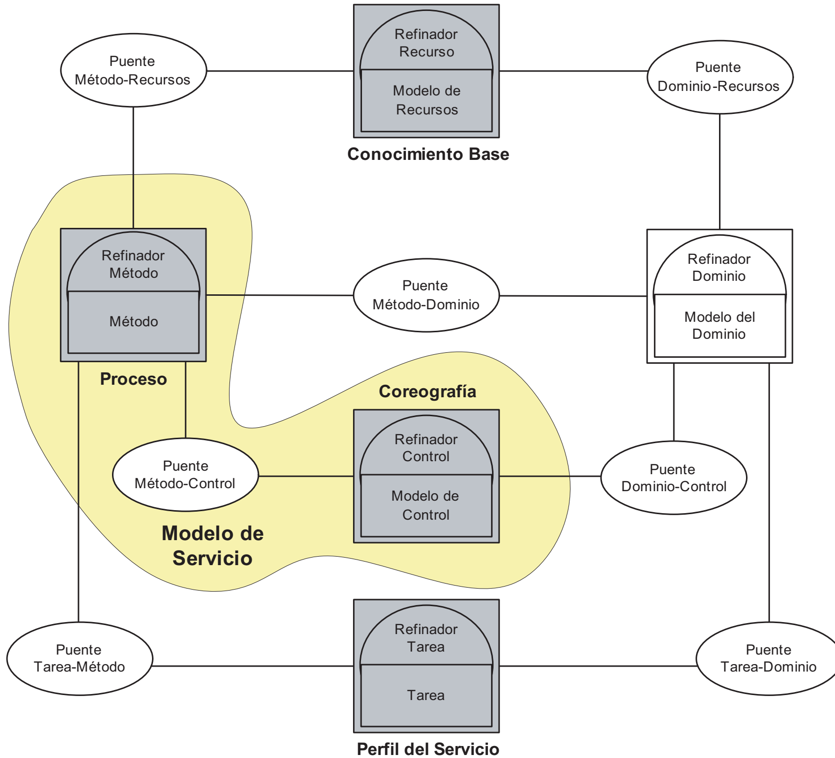


Figura 9.1: Integración de los servicios OWL-S dentro del marco de conocimiento

en redes de Petri de alto nivel (HLPNs), tampoco relacionan la lógica descriptiva utilizada para definir los parámetros del servicio OWL-S con el álgebra que soportará su ejecución. Finalmente, remarcar que únicamente en [183] se pueden ejecutar los servicios OWL-S definidos a través de redes de Petri, ya que en las demás propuestas el objetivo es principalmente la verificación de propiedades.

9.2. Integración de OWL-S en el metamodelo de flujos de trabajo

Cada uno de los tres modelos de OWL-S se integra dentro de uno de los componentes de conocimiento que forman parte de nuestro metamodelo de WFs: los perfiles del servicio con las tareas, el modelo de servicio con los métodos y el modelo de control, y el modelo de conexión con los recursos. La Figura 9.1 muestra gráficamente esta integración sobre el metamodelo donde los componentes coloreados en gris representan los elementos de OWL-S integrados.

A lo largo de este apartado, ilustraremos la adaptación de OWL-S a nuestro

metamodelo a través del servicio de autenticación de usuarios `LogInService` que presenta el perfil `LogInProfile`, se desbribe mediante el proceso `LogInProcess` y está soportado por el modelo de conexión `LogInGrounding`:

```
<!-- Service description -->
<service:Service rdf:ID="LogInService">
  <service:presents rdf:resource="#LogInProfile"/>
  <service:describedBy rdf:resource="#LogInProcess"/>
  <service:supports rdf:resource="#LogInGrounding"/>
</service:Service>
```

9.2.1. Integración del perfil del servicio

El perfil del usuario se integra como una tarea dentro del marco de conocimiento propuesto, lo cual aporta varias mejoras al diseño de servicios OWL-S. Por una parte, facilita la reutilización del perfil del servicio al ser independiente del dominio de aplicación: las entradas, salidas, competencia y demás parámetos se describen con la terminología propia de la funcionalidad que trata de resolver y, por ello, su reutilización y aplicación a diferentes dominios es más sencilla.

Esta solución también facilita que varios métodos puedan relacionarse con un mismo perfil y un mismo método pueda tener varios perfiles independientes. De esta forma, se enriquece el proceso de selección del método que vaya a resolver una determinada tarea.

Tal es la similitud entre los conceptos *tarea* y *perfil de servicio* que únicamente difieren en una propiedad: `hasCategory`. OWL-S modela las categorías a través del concepto `ServiceCategory` que permite clasificar un servicio en múltiples clases. Esta clasificación no se basa en ningún estándar sino que se deja en manos del proveedor. En nuestro metamodelo, la categorización de los servicios se fundamenta en la taxonomía de clasificación UNSPSC[1].

Finalmente, cabe resaltar que algunas de las propiedades que describen el perfil en OWL-S no se han incluido en el concepto `ServicePofile`, ya que en nuestra opinión, están más cerca de la implementación del servicio que de su descripción. Por ejemplo, propiedades como los parámetros locales implican que el proceso tiene una cierta estructura y, por lo tanto, pertenecen a la descripción operacional del método. Otros parámetros como la asociación de un proceso al perfil del servicio carecen de sentido, ya que una de las ventajas de nuestra aproximación consiste en desligar el perfil del proceso que lo implementa.

9.2.2. Integración del modelo de servicios

El objetivo del modelo de servicio de OWL-S es la descripción de la coreografía del servicio a través de los procesos que participan en su ejecución. Para capturar esta coreografía, la especificación OWL-S modela los servicios en base a tres tipos de

[1] http://www.unspsc.org/

procesos: simples, atómicos y compuestos. La integración del modelo de servicio consistirá, por lo tanto, en asociar a cada uno de estos tipos de procesos un componente del metamodelo de WFs:

- *Procesos simples*. Los procesos simples representan servicios abstractos cuya resolución puede ser a través de un proceso atómico o compuesto. Por lo tanto, no tienen una semántica de ejecución propia sino que la toman de los otros dos tipos de procesos. Permiten con ello introducir cierto grado de no determinismo a la especificación de un servicio, y que sea el ejecutor el que se encargue de seleccionar el tipo de proceso que implementa el proceso simple. Por los motivos anteriores, los procesos simples se representan como adaptadores entre los *métodos* y el *control*.

- *Procesos atómicos*. Estos procesos representan servicios que no tienen (sub)-procesos y que se ejecutan en un único paso de ejecución. El metamodelo facilita el modelado de procesos atómicos a través de métodos primitivos que, como ya se señaló en el Capítulo 4, representan procesos que no se descomponen y, por lo tanto, no tienen descripción operacional. El metamodelo delega la ejecución de estos métodos a los recursos del modelo de organización, de forma que cuando un recurso está asociado a un método primitivo entonces ejecutará la operación del servicio.

- *Procesos compuestos*. Estos procesos tienen (sub)procesos y no se pueden resolver en un único paso de ejecución. Se corresponden con los métodos compuestos del metamodelo de WFs y, por lo tanto, tienen asociado un modelo de control con la coreografía del proceso. Para limitar esta coreografía, el control únicamente aceptará comportamientos definidos en OWL-S, es decir, patrones de WFs que representen alguna construcción de OWL-S. Aunque la mayoría de estas estructuras de control están soportadas por alguno de los patrones de WFs descritos en el Capítulo 5, unas pocas tienen asociado un comportamiento que no se ajusta a ninguno de los patrones o que difiere en alguna característica. Para estos casos, se ha completado la capa de patrones de WFs del metamodelo con nuevos patrones.

En lo que resta de este apartado detallaremos las estructuras de control que facilitan la coreografía de los procesos compuestos en OWL-S [277, 269, 163]. Como ya hemos señalado, muchos comportamientos tienen una correspondencia directa con alguno de los patrones de WFs mientras que algunos son propios de OWL-S.

9.2.2.1. Procesos en OWL-S

El control que establece un proceso en OWL-S es similar al de un método de resolución de problemas. Sin embargo, existe una importante diferencia: los servicios OWL-S no tienen postcondición. En su lugar definen un concepto denominado *resultado*, que se asocia a la salida del WF. Por este motivo, aunque la estructura

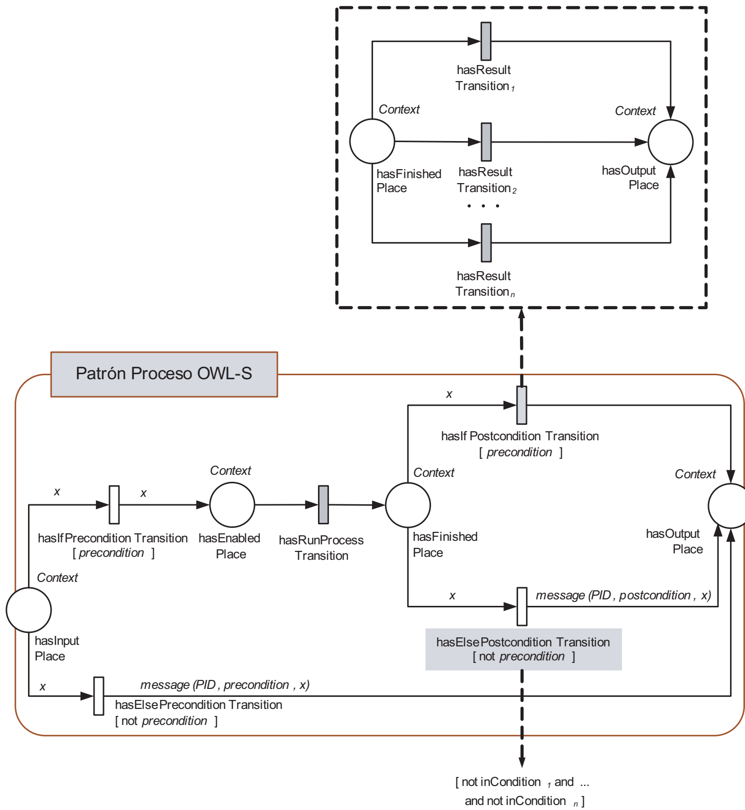


Figura 9.2: Patrón utilizado para la representación de un proceso en OWL-S

del patrón *proceso* vista en el Capítlo 5 es correcta, es necesario puntualizar algunas diferencias. La Figura 9.2 muestra los cambios respecto al patrón original:

1. La transición `hasIfPostconditionTransition` del método, que verifica la postcondición, se sustituye por un conjunto de ramas que tratan las posibles salidas del servicio en función de su ejecución y del estado del entorno. El conjunto de ramas se modelan con un patrón de comportamiento propio de OWL-S denominado *choice*.

2. La rama donde el patrón proceso verifica el no cumplimiento de la postcondición, se modifica para permitir comprobar que no se cumple ninguna de las ramas de resultados. En este caso, la modificación sólo afecta al término que anota la guardia de la transición `hasElsePostconditionTransition`

Con estos cambios, el comportamiento del patrón proceso es el siguiente:

- Un proceso estará activo cuando se cumplan sus precondiciones, es decir, cuando el término precondición de la transición `hasIfPostconditionTransition` se verifique.
- La transición `hasRunProcessTransition` se encargará de realizar la ejecución del proceso.
- Las transiciones `hasResultTransition`$_i$ controlarán los efectos y salidas del proceso en función del contexto en el que se esté ejecutando.

Además, fue necesario añadir al modelo de control un nuevo tipo de página y un nuevo tipo de sustitución:

```
allValuesFrom(hasPages, [ProcessPattern, ResultPattern, ControlPattern]).
allValuesFrom(hasSubstitutions, [ExecuteSubstitution,
  ResultSubstitution, ControlSubstitution]).
```

En este caso, cada una de las páginas del tipo `ResultPattern` del modelo de control se encargará de producir un posible resultado del proceso y las sustituciones del tipo `ResultSubstitution` de unir la red que produce el resultado con la transición *hasResultTransition*$_i$ del patrón proceso que representa. De una forma más precisa, el concepto `ResultSubstitution` se define de la siguiente forma:

```
subClassOf(ResultSubstitution, TransitionSubstitution).

allValuesFrom(hasRefinement, ResultSubstitution, ResultPattern).
```

Este concepto restringe a la relación `hasRefinement` para que únicamente pueda tomar un valor del tipo `ResultPattern`, es decir, un patrón *resultado*.

Aplicado a un problema en particular, la estructura del patrón representado en la Figura 9.2 debe reescribirse adecuándose a las características de cada proceso. Esto significa *(i)* adaptar la estructura del patrón a las características del WF, y *(ii)* adaptar la anotación de la red. Por ejemplo, la Figura 9.3 muestra un WF que representa un proceso de autenticación de usuarios. Se pueden apreciar dos cambios respecto al patrón original: el primero es que la red tiene una única rama de resultados que se encargará de crear una sesión de usuario en el entorno cuando se haya autenticado correctamente; y el segundo cambio está en los términos que anotan la red, como por ejemplo, la condición de guardia asociada a la precondición del proceso.

9.2.2.2. Procesado de resultados en OWL-S

La ejecución de un proceso en OWL-S suele *(i)* transformar la información de entrada (salidas del proceso) y *(ii)* cambiar el estado del mundo (efectos del proceso). OWL-S define el concepto `Result` para referirse a la dupla definida por la salida y el efecto, y con ello facilita la definición de comportamientos diferentes dependiendo

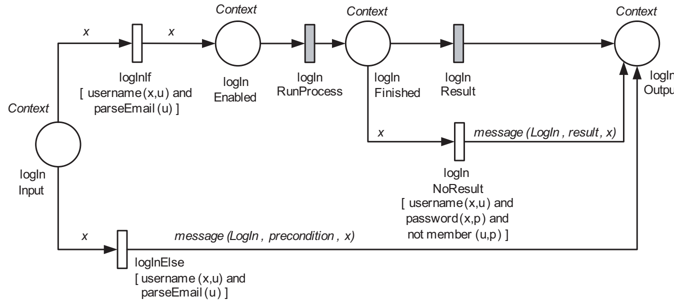


Figura 9.3: Ejemplo de aplicación del patrón *proceso* para un proceso de autenticación de usuarios

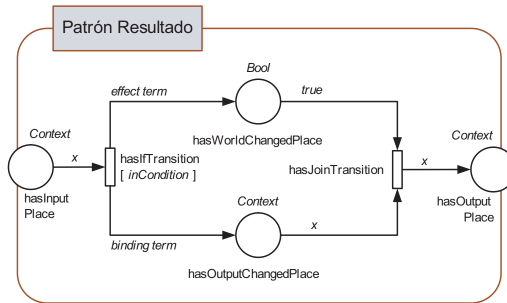


Figura 9.4: Red de Petri que define las salidas y efectos de un proceso en función del contexto de ejecución

del contexto en el cual se ejecuta el proceso. Así, este concepto se puede usar para gestionar la ejecución normal y los fallos del proceso definiendo una rama *resultado* con dicha finalidad. La Figura 9.4 modela este comportamiento: si la precondición de la transición `hasIfTransition` se verifica, los dos arcos de salida de dicha transición realizarán los cambios en la salida del proceso y los efectos en el entorno. El concepto `ResultPattern` captura esta idea:

```
subClassOf(ResultPattern, HLPN).
```

El concepto `ResultPattern` es una `HLPN` que restringe el valor de sus nodos, arcos y anotaciones para cumplir con el patrón *resultado*. Para ello, incorporan una serie de relaciones que identifican cada uno de los nodos del patrón. Los arcos, y con ello la estructura de la red, se verificarán a través de los axiomas recogidos en la Tabla 9.1. Éstas son las principales características de las relaciones definidas por este concepto:

- Relación `hasInputPlace`. Se refiere a la plaza de entrada del patrón *resultado*:

  ```
  domain(hasInputPlace, ResultPattern).
  ```

Tabla 9.1: Axiomas que restringen la semántica del patrón *resultado*

Descripción	Axioma
Existe un único arco entre la plaza de entrada y la transición `hasIfTransition`, y este arco está anotado con una variable.	$\forall R, P, T\ ResultPattern(R) \wedge hasInputPlace(R, P) \wedge hasIfTransition(R, T) \rightarrow uniqueArc(P, T) \wedge variableAnnotated(P, T)$
Existe un único arco entre la plaza `hasWorldChangedPlace` y la transición `hasJoinTransition`, y este arco está anotado con la constante *true*.	$\forall R, P, T\ ResultPattern(R) \wedge hasWorldChangedPlace(R, P) \wedge hasJoinTransition(R, T) \rightarrow uniqueArc(P, T) \wedge operatorAnnotated(P, T, true, [])$
La plaza `hasWorldChangedPlace` tiene el color *boolean*.	$\forall R, P\ ResultPattern(R) \wedge hasWorldChangedPlace(R, P) \rightarrow hasSort(P, boolean)$
Existe un único arco entre la plaza `hasOutputChangedPlace` y la transición `hasJoinTransition`.	$\forall R, P, T\ ResultPattern(R) \wedge hasOutputChangedPlace(R, P) \wedge hasJoinTransition(R, T) \rightarrow uniqueArc(P, T)$
Existe un único arco entre la transición `hasIfTransition` y la plaza `hasWorldChangedPlace`, y este arco está anotado con una llamada al operador *effect*.	$\forall R, P, T\ ResultPattern(R) \wedge hasWorldChangedPlace(R, P) \wedge hasIfTransition(R, T) \rightarrow uniqueArc(T, P) \wedge operatorAnnotated(T, P, effect, [String, Context])$
Existe un único arco entre la transición `hasIfTransition` y la plaza `hasOutputChangedPlace` y con una llamada al operador *withOutput*.	$\forall R, P, T\ ResultPattern(R) \wedge hasOutputChangedPlace(R, P) \wedge hasIfTransition(R, T) \rightarrow uniqueArc(T, P) \wedge operatorAnnotated(T, P, withOutput, [String, Context])$
Existe un único arco entre la transición `hasJoinTransition` y la plaza de salida.	$\forall R, P, T\ ResultPattern(R) \wedge hasOutputPlace(R, P) \wedge hasJoinTransition(R, T) \rightarrow uniqueArc(T, P)$

La red del patrón *resultado* se usa para precisar un efecto en el entorno y en la salida en función del contexto en el que se ejecuta el proceso. Por lo tanto, se creará una instancia de esta red para cada una de las transiciones `hasResultTransitions` definidas en un proceso, es decir, para cada efecto/acción. En esta sustitución es necesario que las interfaces de este patrón sean compatibles con los nodos de entrada y salida de la transición `hasResultTransitions` a sustituir. Ahora bien, al estar todas las plazas del modelo de control anotadas con el mismo color, esta condición siempre se verifica.

- Relación `hasOutputPlace`. Permite identificar la plaza de salida del patrón *resultado*:

```
domain(hasOutputPlace, ResultPattern).
```

Al igual que en el caso de la plaza de entrada, esta plaza también debe tener el mismo tipo que la plaza de salida del patrón *proceso* que es sustituida por la red *resultado*.

- Relación `hasWorldChangedPlace`. Permite identificar una plaza que contendrá valores con el color *boolean*, ya que su función es simplemente indicar que el cambio en el entorno ya ha sido efectuado:

  ```
  objectProperty(hasWorldChangedPlace).
  domain(hasWorldChangedPlace, ResultPattern).
  range(hasWorldChangedPlace, Place).
  cardinality(hasWorldChangedPlace, 1).
  ```

  En realidad esta plaza únicamente almacenará marcas con el valor *true*, y por este motivo el arco que se dirige desde esta plaza a la transición `hasJoinTransition` está anotada por esta constante.

- Relación `hasOutputChangedPlace`. Apunta a una plaza cuyo objetivo es almacenar la transformación de la salida motivada por el contexto en el que se está ejecutando el proceso. Por ello, las marcas de esta plaza contendrán el nuevo contexto una vez que haya sido adaptado:

  ```
  objectProperty(hasOutputChangedPlace).
  domain(hasOutputChangedPlace, ResultPattern).
  range(hasOutputChangedPlace, Place).
  cardinality(hasOutputChangedPlace, 1).
  ```

- Relación `hasIfTransition`. Identificar la transición que contiene la condición de activación del *resultado*:

  ```
  domain(hasIfTransition, ResultPattern).
  ```

  Esta transición, junto con los arcos que parten de ella, es el núcleo del patrón *resultado*, ya que permite *(i)* verificar que el entorno cumple unas determinadas condiciones, *(ii)* aplicar los cambios en el entorno, y *(iii)* transformar la salida para adecuarla al contexto en el que se ejecuta el proceso. Es conveniente recordar que cuando tiene lugar el disparo de una transición, sus arcos de salida se encargan de realizar alguna funcionalidad y de generar las nuevas marcas. Por ello, el disparo de la transición `hasIfTransition` aplica los cambios en el mundo cuando una instancia del contexto de ejecución verifica su precondición. Específicamente, la evaluación del arco anotado con el término `effect` realizará esa funcionalidad. Finalmente, la transición `hasIfTransition` transforma la salida del proceso, en función del estado del contexto de ejecución, a través de la evaluación del término *binding* y genera la nueva salida.

- Relación `hasJoinTransition`. Apunta a una transición cuyo objetivo es sincronizar los efectos en el mundo y las transformaciones en la salida:

  ```
  objectProperty(hasJoinTransition).
  domain(hasJoinTransition, ResultPattern).
  range(hasJoinTransition, Transition).
  cardinality(hasJoinTransition, 1).
  ```

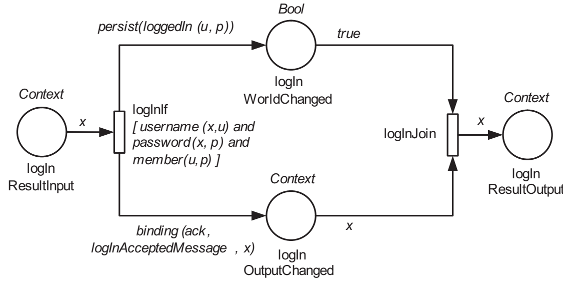


Figura 9.5: Ejemplo de aplicación del patrón resultado para un proceso de autenticación de usuarios

> Una vez finalizadas ambas acciones, las plazas `hasWorldChangedPlace` y `hasOutputChangedPlace` tendrán una marca y la transición `hasJoinTransition` estará activa. La ejecución de esta transición eliminará las marcas de dichas plazas y generará una nueva marca en la plaza `hasOutputPlace`, que será el contexto de ejecución final del proceso.

La Figura 9.5 muestra una posible rama *resultado* para el proceso de autenticación de usuarios visto en la Figura 9.3. En este ejemplo, la condición `username(x,u) and password(x, p) and member(username,password)` se encarga de comprobar si el usuario está registrado en el sistema. En caso afirmativo, la llamada al operador `persist(loggedIn(u, p))` cambiará el estado del usuario a autenticado dentro del sistema. También se realizará la llamada al operador `binding(ack, logInAcceptedMessage, x)` para que la salida del proceso devuelva un mensaje de confirmación al usuario.

9.2.2.3. Secuencia en OWL-S

Permite ejecutar ordenadamente un conjunto de procesos. OWL-S especifica la secuencia a ejecutar a través del atributo `components` de la clase `Sequence` que contiene una lista de estructuras de control. Se utiliza directamente el patrón *secuencia* de los WFs (concepto `SequencePattern`) para representar esta estructura de control. La Tabla 9.2 recoge el axioma que verifica que las secuencias son equivalentes.

9.2.2.4. Separación en OWL-S

Una *separación* en OWL-S describe una estructura donde un conjunto de procesos pueden ejecutarse concurrentemente aunque a diferencia del patrón original de WFs no requiere la finalización de los procesos concurrentes. La incorporación de los (sub)procesos a ejecutar en las ramas concurrentes es otra de las diferencias introducidas en el patrón *separación* de OWL-S. OWL-S se refiere a esta estructura

Tabla 9.2: Axiomas que restringen la *secuencia*

Todo concepto `Sequence` de OWL-S estará asociado a un patrón `SequencePattern`.
$\forall P\ Sequence(P) \rightarrow\ \exists M, R\ OWLSMapping(M) \wedge hasSource(M, P) \wedge hasTarget(M, R) \wedge SequencePattern(R)$
El patrón *secuencia* tiene el mismo número de bloques de control que elementos tiene la propiedad `components` de OWL-S.
$\forall P, M, R, P, N_1, N_2\ Sequence(P) \wedge OWLSMapping(M) \wedge SequencePattern(R) \wedge hasSource(M, P) \wedge hasTarget(M, R) \wedge cardinality(P, components, N_1) \wedge cardinality(S, hasControlList, N_1) \rightarrow N_1 = N_2$

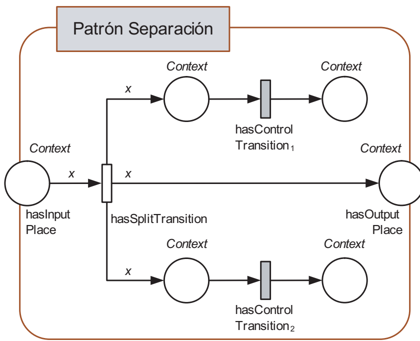


Figura 9.6: Representación del patrón separación

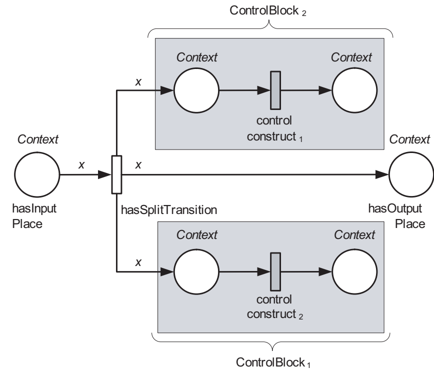


Figura 9.7: Bloques de control del patrón separación

mediante la clase `Split` y establece en su propiedad `components` al multiconjunto de construcciones de control a ejecutar concurrentemente.

El diseño de este patrón parte de la red *separación* vista en el Capítulo 5 (concepto `SplitPattern`) al que extiende *(i)* para finalizar la ejecución del patrón independientemente de las ramas concurrentes y *(ii)* para soportar las construcciones de control concurrentes. El concepto `OWLSSplit` se refiere a este patrón:

```
subClassOf(OWLSSplit, SplitPattern).
```

Como se puede ver en la Figura 9.6, se añade la plaza `hasOutputPlace` para referirse a la plaza de salida del patrón. Además, cada una de las plazas `hasOutputPlaces` de la separación original debe conectarse con una de las transiciones `hasControlTransition`$_i$ del patrón. Esta característica se especifica a través de la relación `hasControlBlocks` que captura el conjunto de ramas concurrentes a través de una instancia del tipo `ControlBlock` visto en el Capítulo 5:

```
objectProperty(hasControlBlocks).
domain(hasControlBlocks, SplitPattern).
```

Tabla 9.3: Axiomas que restringen la *separación*

Todo concepto `Split` de OWL-S estará asociado a un patrón `SplitPattern`.
$\forall P\ Split(P) \rightarrow \exists M, R\ OWLSMapping(M) \wedge hasSource(M,P) \wedge hasTarget(M,R) \wedge OWLSSplit(R)$
El patrón *separación* tiene el mismo número de bloques de control que elementos tiene la propiedad `components` de OWL-S.
$\forall P, M, R, P, N_1, N_2\ Split(P) \wedge OWLSMapping(M) \wedge OWLSSplit(R) \wedge hasSource(M,P) \wedge hasTarget(M,R) \wedge cardinality(P, components, N_1) \wedge cardinality(S, hasControlBlocks, N_1) \rightarrow N_1 = N_2$
Existe un único arco entre la transición `hasSplitTransition` y la plaza de salida y este arco está anotado con una variable.
$\forall R, P, T\ OWLSSplit(R) \wedge Place(P) \wedge Transition(T) \wedge hasOutputPlace(R,P) \wedge hasSplitTransition(R,T) \rightarrow uniqueArc(T,P) \wedge variableAnnotated(T,P)$
La plaza de entrada del bloque de control se corresponde con una de las plazas de salida de la red de *separación*.
$\forall R, P_1, T\ OWLSSplit(R) \wedge Place(P_1) \wedge hasOutputPlace(R,P_1) \rightarrow \exists\ B, P_2 \wedge hasControlBlocks(R,B) \wedge hasInputPlace(B,P_2) \wedge P_1 = P_2$

```
range(hasControlBlocks, ControlBlock).
minCardinality(hasControlBlocks, 1).
```

Por lo tanto, la entrada de cada bloque será una de las plazas de salida del patrón *separación* original. Los axiomas que complementan la semántica de este patrón están descritos en la Tabla 9.3.

9.2.2.5. Elección en OWL-S

Al contrario del patrón *elección exclusiva* original, que busca la elección de una rama de entre un conjunto de ramas seleccionables, la estructura en OWL-S identifica cada una de estas ramas con un proceso. La clase `Choice` da soporte a esta estructura en OWL-S y define al conjunto de procesos seleccionables como un multiconjunto de estructuras de control. Otra diferencia está en la finalización del patrón: en OWL-S todas las ramas seleccionables se mezclan en el mismo punto de ejecución, es decir, todas las ramas compartirán la plaza de salida tal y como se puede ver en la Figura 9.8:

```
subClassOf(OWLSChoice, ChoicePattern).
```

Los axiomas de la Tabla 9.4 garantizan el comportamiento de este patrón.

9.2.2.6. Patrón separación-unión

Esta estructura de control describe la ejecución concurrente de un conjunto de procesos entre dos puntos de control. OWL-S captura este comportamiento a través de

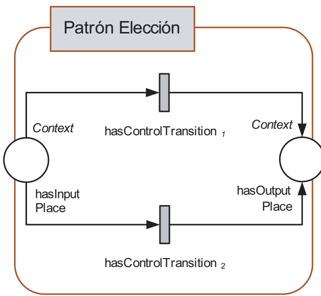


Figura 9.8: Representación del patrón elección

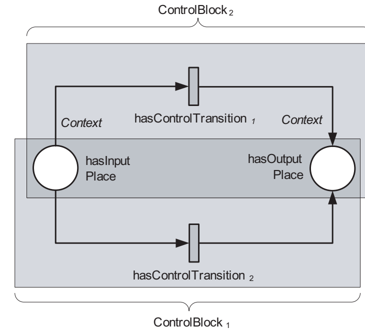


Figura 9.9: Bloques de control del patrón elección

la clase `SplitJoin` e indica a través de la relación `components` al multiconjunto de procesos que pueden ejecutarse concurrentemente. El patrón de WFs denominado `SplitJoinPattern` descrito en el Capítulo 5 captura con exactitud este comportamiento por lo que una instancia `Split-Join` de OWL-S se corresponderá con un patrón `SplitJoinPattern` de WFs. Los axiomas de la Tabla 9.5 completan la semántica de esta asociación.

9.2.2.7. Patrón sin-orden

Esta estructura de control permite la ejecución de un conjunto de procesos sin un orden predeterminado y no concurrentemente. La ejecución y finalización de cada uno de los componentes de esta estructura es obligatoria. La clase `Any-Order` de OWL-S captura esta estructura de control y tiene asociada a través de la relación `components` al multiconjunto de procesos que se pueden ejecutar concurrentemente. Al contrario de las estructuras de control anteriores, ésta no tiene una correspondencia directa con ningún patrón de WFs, lo cual hace que sea necesario añadirla a nuestra capa de patrones. La Figura 9.10 representa la HLPN de este patrón en la cual se pueden apreciar tres comportamientos: *separación*, *unión* y *elección*. En este caso, las transiciones concurrentes de la *separación* se conectan al punto de exclusión mutua definido por la plaza `hasOrSplitPlace`. Esta plaza contendrá una única marca, ya que sino habilitaría la ejecución de más de una de las ramas concurrentemente. Por lo tanto, cuando uno de los procesos se ejecuta, éste elimina la marca de la plaza `hasOrSplitPlace` y desactiva a las otras ramas. Cuando el proceso finaliza su ejecución, se crea una nueva marca en la plaza `hasOrSplitPlace` y se activan nuevamente las demás ramas. El concepto `OWLSAnyOrder` se refiere a este patrón:

```
subClassOf(OWLSAnyOrder, WFPattern).

domain(hasSplit, OWLSAnyOrder).
domain(hasJoin, OWLSAnyOrder).
```

Tabla 9.4: Axiomas que restringen la *elección*

Si A es el conjunto de bloques de control y B es la lista de componentes del concepto `Choice` de OWL-S, $\|A\| = \|B\|$.
$\forall S, P, N_1, N_2\ OWLSChoice(S) \wedge Choice(P) \wedge hasControlConstruct(S, P) \wedge cardinality(S, hasControlBlocks, N_1) \wedge cardinality(P, components, N_2) \rightarrow N_1 = N_2$
La plaza de salida del patrón es la misma que la plaza de salida de cada una de las ramas de control.
$\forall S, B, P\ OWLSChoice(S) \wedge ControlBlock(B) \wedge hasControlBlocks(S, B) \wedge hasOutputPlace(S, P) \rightarrow hasOutputPlace(B, P)$

Tabla 9.5: Axiomas que restringen la *separación-unión*

Todo concepto `SplitJoin` de OWL-S estará asociado a un patrón `SplitJoinPattern`.
$\forall P\ SplitJoin(P) \rightarrow\ \exists M, R\ OWLSMapping(M) \wedge hasSource(M, P) \wedge hasTarget(M, R) \wedge SplitJoinPattern(R)$
El patrón *separación-unión* tiene el mismo número de ramas paralelas que la propiedad `components` de OWL-S.
$\forall P, M, R, P, N_1, N_2\ SplitJoin(P) \wedge OWLSMapping(M) \wedge SplitJoinPattern(R) \wedge hasSource(M, P) \wedge hasTarget(M, R) \wedge cardinality(P, components, N_1) \wedge cardinality(S, hasControlBlocks, N_1) \rightarrow N_1 = N_2$

Dada la complejidad de este patrón se puede describir en base a cada uno de los patrones que lo componen. En este caso, la descripción se centrará en las partes de separación, de unión y de elección:

- *Separación.* Se identifica a través de la relación `hasSplit` que se refiere a un patrón de *separación*. Por lo tanto, tendrá una plaza de entrada y una transición desde la cual partan cada una de las ramas concurrentes. Uno de los arcos irá dirigido a la plaza `hasOrSplitPlace` que actúa como punto de conflicto entre las ramas concurrentes:

```
objectProperty(hasOrSplitPlace).
domain(hasOrSplitPlace, OWLSAnyOrder).
range(hasOrSplitPlace, AnyBlock).
cardinality(hasOrSplitPlace, 1).
```

- *Sincronización.* Cada una de las ramas concurrentes y la plaza `hasOrSplitPlace` se sincronizan en un único punto de la HLPN. La relación `hasJoin` se refiere al patrón *sincronización* a cargo de este comportamiento: todas las transiciones de salida de cada una de las ramas concurrentes se conectan a la transición `hasJoinTransition` y su ejecución generará una marca en la plaza de salida de la red.

- *Ramas concurrentes.* El patrón *sin-orden* captura las ramas concurrentes a través de la relación `hasAnyBlocks`:

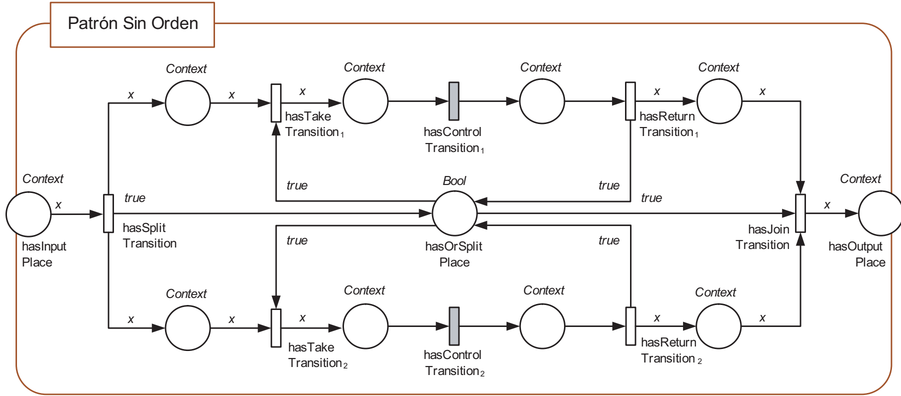


Figura 9.10: Ejecución sin un orden predeterminado de dos procesos

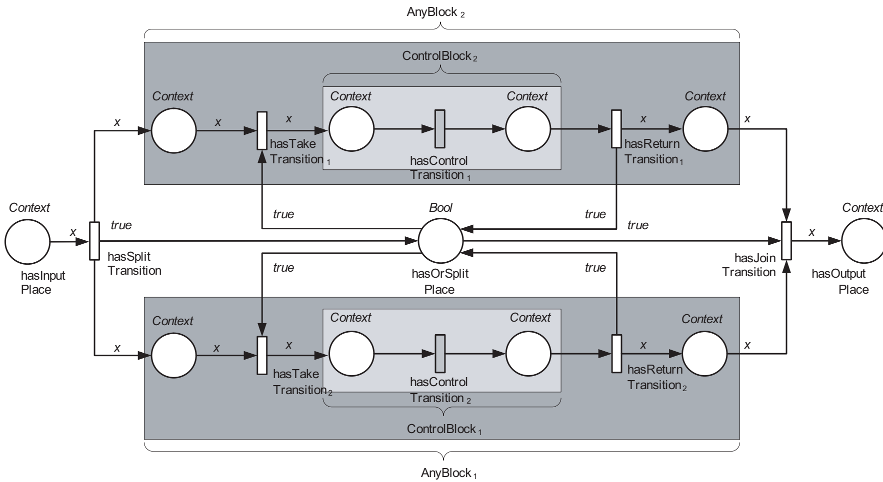


Figura 9.11: Bloques de control del patrón sin-orden

```
objectProperty(hasAnyBlocks).
domain(hasAnyBlocks, OWLSAnyOrder).
range(hasAnyBlocks, AnyBlock).
minCardinality(hasAnyBlocks, 1).
```

Cada una de estas ramas está modelada a través de la clase `AnyBlock`:

```
subClassOf(AnyBlock, Thing).

domain(hasInputPlace, AnyBlock).
domain(hasOutputPlace, AnyBlock).

objectProperty(hasControlBlock).
```

Tabla 9.6: Axiomas que restringen las ramas concurrentes del patrón *sin-orden*

Existe un único arco entre la plaza de entrada de la rama y la transición `hasTakeTransition` y este arco está anotado con una variable.
$\forall R, P, T\ AnyBlock(R) \wedge hasInputPlace(R, P) \wedge hasTakeTransition(R, T) \rightarrow uniqueArc(P, T) \wedge variableAnnotated(P, T)$
Existe un único arco entre la transición `hasReturnTransition` y la plaza de salida del patrón y este arco está anotado con una variable.
$\forall R, P, T\ AnyBlock(R) \wedge hasOutputPlace(R, P) \wedge hasReturnTransition(R, T) \rightarrow uniqueArc(T, P) \wedge variableAnnotated(T, P)$
Existe un único arco entre la transición `hasTakeTransition` y la plaza de entrada de la rama de control y este arco está anotado con una variable.
$\forall R, P, T\ AnyBlock(R) \wedge hasControlBlock(R, B) \wedge hasInputPlace(B, P) \wedge hasTakeTransition(R, T) \rightarrow uniqueArc(T, P) \wedge variableAnnotated(T, P)$
Existe un único arco entre la plaza de salida de la rama de control y la transición `hasReturnTransition` y este arco está anotado con una variable.
$\forall R, P, T\ AnyBlock(R) \wedge hasControlBlock(R, B) \wedge hasOutputPlace(B, P) \wedge hasReturnTransition(R, T) \rightarrow uniqueArc(P, T) \wedge variableAnnotated(P, T)$

```
domain(hasControlBlock, AnyBlock).
range(hasControlBlock, ControlBlock).
cardinality(hasControlBlock, 1).
```

En la Figura 9.11 se ensombrecen en gris oscuro las instancias del concepto `AnyBlock` sobre el patrón *sin-orden*. Las relaciones `hasInputPlace` y `hasOutputPlace` de este concepto hacen referencia respectivamente a la plaza de entrada y a la plaza de salida de la rama mientras que la relación `hasTakeTransition` apunta a la transición que competirá por los recursos compartidos:

```
objectProperty(hasTakeTransition).
domain(hasTakeTransition, AnyBlock).
range(hasTakeTransition, Transition).
cardinality(hasTakeTransition, 1).
```

Finalmente, la relación `hasReturnTransition` se refiere a la transición que libera los recursos compartidos:

```
objectProperty(hasReturnTransition).
domain(hasReturnTransition, AnyBlock).
range(hasReturnTransition, Transition).
cardinality(hasReturnTransition, 1).
```

Los axiomas de la Tabla 9.6 completan la definición de las ramas de control del patrón *sin-orden*.

- *Conflicto*. Hasta este momento, cada una de las ramas del patrón ha sido descrita como si fuese independiente y concurrente con las demás. Para evitar la

Tabla 9.7: Axiomas que restringen el patrón *sin-orden*

Existe un único arco entre la transición `hasSplitTransition` y la plaza `hasOrSplitPlace` y este arco está anotado con la constante *true*.
$\forall R, P, T\ OWLSAnyOrder(R) \wedge Place(P) \wedge hasOrSplitPlace(R, P) \wedge hasSplitTransition(R, T) \rightarrow uniqueArc(T, P) \wedge operatorAnnotated(T, P, true, [])$
Existe un único arco entre la plaza `hasOrSplitPlace` y la transición `hasJoinTransition` y este arco está anotado con la constante *true*.
$\forall R, P, T\ OWLSAnyOrder(R) \wedge Place(P) \wedge hasOrSplitPlace(R, P) \wedge hasJoinTransition(R, T) \rightarrow uniqueArc(P, T) \wedge operatorAnnotated(P, T, true, [])$

Tabla 9.8: Axiomas para la integración de la separación y sincronización con cada una de las ramas concurrentes

Todas las plazas de salida de la separación, a excepción de la plaza en conflicto, son entrada de alguna rama concurrente.
$\forall R, P, T\ OWLSAnyOrder(R) \wedge hasSplit(S) \wedge hasOutputPlaces(S, P) \wedge nothasOrSplitPlace(R, P) \rightarrow \exists B\ hasAnyBlocks(R, B) \wedge hasInputPlace(B, P)$
Todas las plazas de entrada de la sincronización, a excepción de la plaza en conflicto, son salida de alguna rama concurrente.
$\forall R, P, T\ OWLSAnyOrder(R) \wedge hasJoin(S) \wedge hasInputPlaces(S, P) \wedge nothasOrSplitPlace(R, P) \rightarrow \exists B\ hasAnyBlocks(R, B) \wedge hasOutputPlace(B, P)$
Existe un único arco entre cada transición *return flag* de las ramas concurrentes y la plaza `hasOrSplitPlace` y este arco está anotado con la constante true.
$\forall R, P, T\ OWLSAnyOrder(R) \wedge hasAnyBlocks(R, B) \wedge hasOrSplitPlace(R, P) \wedge hasReturnTransition(B, T) \rightarrow uniqueArc(T, P) \wedge operatorAnnotated(T, P, true, [])$
Existe un único arco entre la plaza `hasOrSplitPlace` y la transición `hasTakeTransition` de cada una de las ramas concurrentes y este arco está anotado con la constante true.
$\forall R, P, T\ OWLSAnyOrder(R) \wedge hasAnyBlocks(R, B) \wedge hasOrSplitPlace(R, P) \wedge hasTakeTransition(B, T) \rightarrow uniqueArc(P, T) \wedge operatorAnnotated(P, T, true, [])$

concurrencia entre las distintas ramas, la plaza `hasOrSplitPlace` actúa como recurso compartido. Las transiciones `hasTakeTransition` de cada rama concurrente competirán por conseguir dicho recurso y su ejecución eliminará la marca en la plaza de entrada y de la plaza compartida. Este evento provocará que *(i)* esta rama no se pueda volver a ejecutar y *(ii)* que ninguna de las otras ramas del patrón puedan iniciar su ejecución. Una vez bloqueadas las demás ramas, el siguiente paso consiste en ejecutar la parte de control de la rama activada.

Una vez finalizada la ejecución de la parte de control, la transición `hasReturnTransition` se encarga de realizar el proceso contrario. Por un lado, finaliza la ejecución de la rama generando una nueva marca en la plaza de salida. Por el otro, restituye el recurso compartido en la plaza `hasOrSplitPlace` de forma que las demás ramas vuelvan a estar activas.

Los axiomas de las tablas 9.7 y 9.8 completan la definición de este patrón.

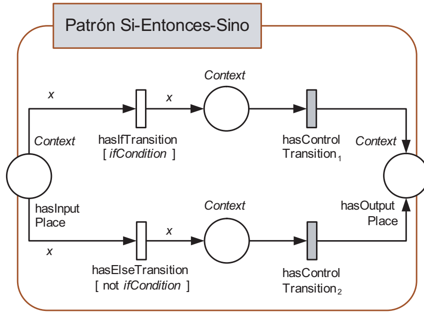


Figura 9.12: Representación del patrón if-then-else

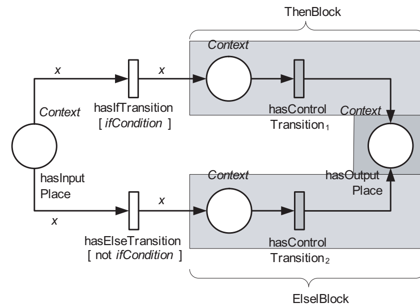


Figura 9.13: Bloques de control del patrón if-then-else

9.2.2.8. Patrón si-entonces-sino

Esta construcción describe la típica estructura de control de los lenguajes de programación donde la evaluación de una condición activa la ejecución de la rama *entonces* o de la rama *sino*. OWL-S define la clase `If-Then-Else` para dar soporte a esta estructura y le asocia las propiedades `ifCondition`, `then` y `else` para almacenar la condición y la construcción de control asociada a las ramas *entonces* y *sino* respectivamente. No existe un patrón de WFs que soporte directamente esta construcción pero sí combinando los patrones *elección exclusiva* y mezcla. La Figura 9.12 muestra la red asociada a este patrón de comportamiento. En este caso, se han añadido las precondiciones de cada una de las ramas del patrón *elección exclusiva*. El concepto `IfThenElsePattern` se refiere a este patrón:

```
subClassOf(IfThenElsePattern, ChoiceMergePattern).

domain(hasIfTransition, IfThenElsePattern).
domain(hasElseTransition, IfThenElsePattern).
```

Para facilitar la identificación de las ramas *entonces* y *sino* se han añadido las relaciones `hasIfTransition` y `hasElseTransition`. La transición `hasIfTransition` tendrá como condición de guardia al término `ifCondition` mientras que la transición `hasElseTransition` tendrá como precondición a ese mismo término pero negado. Así, las dos transiciones nunca estarán activas al mismo tiempo.

De la misma forma también se identificaron los bloques de control de las ramas *entonces* y *sino* pertenecientes a la *mezcla*:

```
objectProperty(hasThenBlock).
domain(hasThenBlock, IfThenElsePattern).
range(hasThenBlock, ControlBlock).
cardinality(hasThenBlock, 1).
```

Tabla 9.9: Axiomas que restringen la rama *si* del patrón *si-entonces-sino*

La elección sólo tiene dos plazas de salida.
$\forall R, P, T\ IfThenElsePattern(R) \wedge hasChoicePattern(R, S) \wedge cardinality(S, hasOutputPlaces, N) \rightarrow N = 2$
La mezcla sólo tiene dos plazas de entrada.
$\forall R, P, T\ IfThenElsePattern(R) \wedge hasMergePattern(R, S) \wedge cardinality(S, hasInputPlaces, N) \rightarrow N = 2$
La transición `hasIfTransition` está anotada con la condición de guardia *ifCondition* de OWL-S.
$\forall P, M, R, C\ IfThenElse(P) \wedge OWLSMapping(M) \wedge IfThenElsePattern(R) \wedge hasSource(M, P) \wedge hasTarget(M, R) \wedge hasIfTransition(R, T) \wedge ifCondition(P, C) \rightarrow hasGuard(T, C)$
La transición `hasElseTransition` está anotada con la condición de guardia de la transición `hasIfTransition` negada.
$\forall P, T_1, T_2, G_1, G_2\ IfThenElsePattern(P) \wedge hasIfTransition(P, T_1) \wedge hasElseTransition(P, T_2) \wedge hasGuard(T_1, G_1) \rightarrow hasGuard(T_2, G_2) \wedge hasOperator(G_2, boolean_not) \wedge hasArguments(G_2, L) \wedge member(G_1, L)$

```
objectProperty(hasElseBlock).
domain(hasElseBlock, IfThenElsePattern).
range(hasElseBlock, ControlBlock).
cardinality(hasElseBlock, 1).
```

La Figura 9.13 representa gráficamente el bloque *entonces* y el bloque *sino* de este patrón, que, como se puede apreciar, comparten la plaza de salida. Éste y otros axiomas descritos en la Tabla 9.9 completan la semántica de este patrón.

9.2.2.9. Patrón repetir-mientras

OWL-S define bucles *repetir-mientras* a través de la clase `Repeat-While`. Esta clase extiende la clase `ControlConstruct` donde las propiedades `whileCondition` y `whileProcess` se usan para establecer respectivamente la condición y la construcción de control asociada al cuerpo del bucle. Este patrón de OWL-S se implementa con el patrón iterativo de WFs denominado *repetir-mientras* y descrito en el Capítulo 5. Las correspondencias entre la definición de OWL-S y el patrón de WFs están descritas en la Tabla 9.10.

9.2.2.10. Patrón repetir-hasta

Esta estructura de control describe un bucle *repetir-hasta*: se ejecuta el cuerpo del bucle hasta que se cumpla una condición de salida. OWL-S define la clase `Repeat-Until` para dar soporte a esta construcción. Esta clase extiende la clase `ControlConstruct`

Tabla 9.10: Axiomas que restringen al patrón *repetir-mientras*

La transición `hasIfTransition` tiene como condición de guardia a la condición *whileCondition* del concepto `Repeat-While` de OWL-S que representa.
$\forall P, M, R, C\ RepeatWhile(P) \wedge OWLSMapping(M) \wedge RepeatWhilePattern(R) \wedge hasSource(M, P) \wedge hasTarget(M, R) \wedge hasIfTransition(R, T) \wedge whileCondition(P, C) \rightarrow hasGuard(T, C)$

Tabla 9.11: Axiomas que restringen al patrón *repetir-hasta*

La transición `hasIfTransition` tiene como condición de guardia a la condición *untilCondition* del concepto `Repeat-Until` de OWL-S que representa.
$\forall P, M, R, C\ RepeatUntil(P) \wedge OWLSMapping(M) \wedge RepeatUntilPattern(R) \wedge hasSource(M, P) \wedge hasTarget(M, R) \wedge hasIfTransition(R, T) \wedge untilCondition(P, C) \rightarrow hasGuard(T, C)$

con las propiedades `untilCondition` y `untilProcess` mediante las cuales define la condición y la construcción de control asociada al cuerpo del bucle. Las correspondencias entre la definición de OWL-S y el patrón de WFs están descritas en la Tabla 9.11.

9.2.3. Integración del modelo de conexión del servicio

El modelo de conexión del servicio OWL-S detalla cómo el agente puede invocar al servicio, e incluye al protocolo, formato de los mensajes, serialización, transporte y direccionamiento. El propósito del modelo de conexión de OWL-S es mostrar como las entradas y salidas de un proceso atómico se pueden transformar en un mensaje utilizando el Web Services Description Language (WSDL) [69] que posee mecanismos para realizar dicha transformación. Aunque las correspondencias entre OWL-S y WSDL están fuera del alcance de esta memoria, es necesario destacar dos detalles:

- Los procesos atómicos se corresponden con operaciones WSDL.
- Las entradas y salidas de un proceso atómico se corresponden con mensajes WSDL.

El siguiente código muestra un fragmento del modelo de conexión empleado en el ejemplo de autenticación de usuarios:

```
<!-- Grounding description -->
<grounding:WsdlGrounding rdf:ID="LogInGrounding">
  <service:supportedBy rdf:resource="#LogInService"/>
  <grounding:hasAtomicProcessGrounding rdf:resource="#LogInProcessGrounding"/>
</grounding:WsdlGrounding>

<grounding:WsdlAtomicProcessGrounding rdf:ID="LogInProcessGrounding">
  <grounding:owlsProcess rdf:resource="#LogInProcess"/>
```

```
<grounding:wsdlOperation>
  <grounding:WsdlOperationRef>
    <grounding:portType rdf:datatype="&xsd;#anyURI">
      wsdl_grounding;#LogInPortType
    </grounding:portType>
    <grounding:operation rdf:datatype="&xsd;#anyURI">
      wsdl_grounding;#LogInOperation
    </grounding:operation>
  </grounding:WsdlOperationRef>
</grounding:wsdlOperation>

<grounding:wsdlInputMessage rdf:datatype="&xsd;#anyURI">
  wsdl_grounding;#LogInInput
</grounding:wsdlInputMessage>

<grounding:wsdlInput>
  <grounding:WsdlInputMessageMap>
    <grounding:owlsParameter rdf:resource="#Username"/>
    <grounding:wsdlMessagePart rdf:datatype="&xsd;#anyURI">
      wsdl_grounding;#Username
    </grounding:wsdlMessagePart>
  </grounding:WsdlInputMessageMap>
</grounding:wsdlInput>
...
```

En este ejemplo, el servicio se asocia a la operación WSDL llamada `LogInOperation`. Además, la entrada del servicio que define el nombre de usuario se asocia con el mensaje. Es necesario comentar que este ejemplo muestra un caso sencillo donde el mensaje WSDL se corresponde directamente con la entrada OWL-S. En casos más complejos, un XSLT se encargará de realizar la transformación.

Como señalamos en el apartado 9.2.2 el metamodelo representa a los procesos atómicos, aquellos que tienen una operación WSDL asociada, a través de métodos primitivos. En la dinámica de ejecución de los WFs, un método primitivo no tiene una implementación asociada sino que se delega la ejecución de su funcionalidad a los recursos. Para cada método primitivo existirá un conjunto de recursos habilitados para cubrir su funcionalidad, aunque éstps participan en la ejecución del WF como agentes externos y su implementación es externa al metamodelo.

Por lo tanto, la integración del modelo de conexión se realiza al nivel de implementación de los recursos y no en el metamodelo. En realidad, todos los recursos que participan en la ejecución de un servicio OWL-S tienen la misma estructura y se encargan de *(i)* transformar las entradas del proceso en un mensaje, *(ii)* ejecutar la operación y *(iii)* transformar el mensaje resultado de la operación en las salidas del proceso. Como el recurso sabe qué método va a realizar, simplemente tendrá que recuperar el modelo de conexión asociado de OWL-S.

9.3. Análisis de la coreografía de servicios OWL-S basada en redes de Petri

Al contrario de lo que sucede con otras técnicas de modelado e incluso con otros formalismos, las redes de Petri facilitan el análisis de ciertas propiedades del modelo resultante. Este análisis aplicado al diseño de servicios es de gran utilidad para evitar fallos antes de que los servicios sean puestos en producción. Cabe mencionar que algunos de los algoritmos de análisis son generalmente costosos en tiempo y computación. Sin embargo, aplicado a los servicios OWL-S el efecto es mínimo, ya que son típicamente pequeños.

9.3.1. Validación de servicios OWL-S

La validación es una tarea importante en el desarrollo de los servicios, ya que permite comprobar que un servicio se comporta de la forma esperada. Aunque existen diferentes soluciones al usar redes de Petri, en este desarrollo se ha aplicado la simulación por lotes (*batch simulation* en inglés) para realizar la validación de las redes. Específicamente, se comprueba que la ejecución de un conjunto de casos de prueba genera las salidas esperadas y efectos deseados en el entorno. Claro está que no siempre se puede realizar una validación completa, ya que las variables en OWL-S pueden tener un dominio de datos infinito. Por ello, esta validación se suele realizar únicamente para aquellos casos en los cuales el servicio ha sido desarrollado o aquellos que comprueban sus límites.

También se han analizado los patrones descritos a lo largo de esta sección. Para cada uno de estos patrones se ha compuesto un servicio y simulado su comportamiento. Por ejemplo, se ha validado que el patrón *si-entonces-sino* ejecuta la rama *entonces* cuando se verifica la condición *si*. Cada una de estas simulaciones ha sido realizada sobre el motor de OWL-S y sus resultados han sido analizados para verificar que efectivamente cumplen con la especificación OWL-S.

Cabe recordar nuevamente que OWL-S no especifica de forma completa el comportamiento de algunas de sus construcciones de control ni el desenlace del servicio cuando no se cumplen las precondiciones del proceso, de los resultados o simplemente cuando el servicio no da señales de vida. Por ejemplo, si un proceso incluye una construcción de control del tipo *separación*, OWL-S no requiere que los componentes concurrentes finalicen. Es decir, el servicio puede finalizar antes de la ocurrencia de los procesos en paralelo. En este caso, OWL-S tolera que los servicios en paralelo nunca finalicen y, por lo tanto, la funcionalidad del servicio quede incompleta. De hecho, al contrario de WS-BPEL, OWL-S no especifica el tratamiento de errores, tiempos límite o tolerancia a fallos. En este caso, en OWL-S estricto sería posible que la ejecución de un servicio nunca finalice. Sin embargo, el modelo que se ha presentado a lo largo de esta sección sí soporta el tratamiento de las precondiciones, de los resultados y de los tiempos límite. En este sentido, las simulaciones llevadas a cabo sobre este modelo han demostrado que resuelve varias carencias de la especificación OWL-S en cuanto al tratamiento de errores.

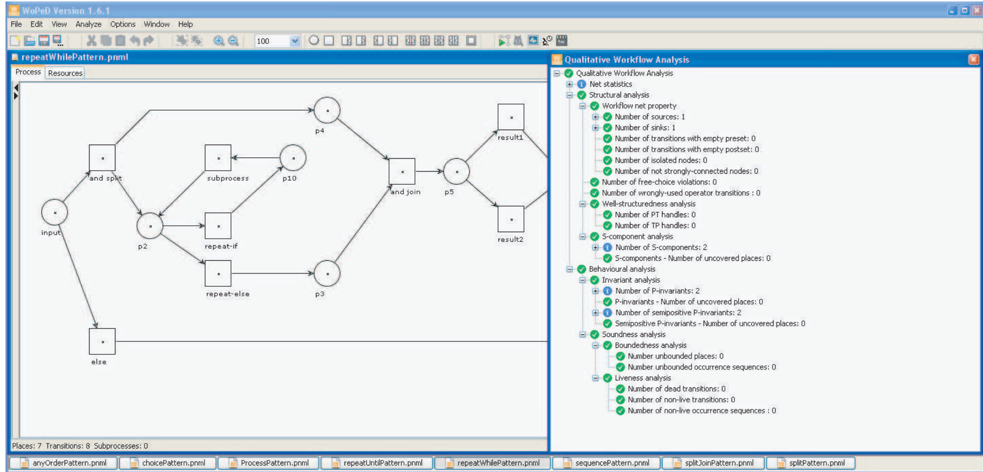


Figura 9.14: Pantallazo de la herramienta WoPeD utilizado para la verificación de las estructuras de control de los patrones *proceso* y *repetir-mientras*

9.3.2. Verificación de servicios OWL-S

Una característica interesante de las redes de Petri, especialmente para el diseño de servicios OWL-S, es su capacidad para detectar errores antes de la puesta en producción de un servicio. Parte importante de esta detección de errores es la verificación de los servicios, donde se establece si el servicio es correcto, es decir, si su modelo cumple con la especificación del servicio. Aplicado a las redes de Petri, un modelo será correcto si cumple con las propiedades establecidas en su especificación.

Los procesos OWL-S modelados en este trabajo tienen una gran expresividad, lo que dificulta su verificación. Ello se debe al uso de colores en las redes, que a efectos de verificación pueden llegar a generar un espacio de estados de dimensión infinita y la consiguiente imposibilidad de llevar a cabo el análisis. Teniendo esto en cuenta, la semántica de verificación empleada abstrae los colores, ya que las variables que anotan las redes pueden tener un dominio infinito. Por lo tanto, la verificación se realiza sobre las redes de bajo nivel, donde las variables se modelan como marcas indistinguibles y las ramas de la red que dependen de los datos (como las condiciones) se modelan mediante plazas que posibilitan elecciones no determísticas.

En la actualidad existen muchas técnicas de verificación disponibles para las redes de Petri, que permiten analizar sus propiedades estructurales o de comportamiento, pero, ya que la especificación OWL-S modela los servicios como WFs, nuestro análisis se limitará a las siguientes propiedades *inicio/fin*, *fuertemente conexa*, *segura*, *sin bloqueos*, *viva*, *bien estructurada* y *libre de conflictos* vistas en el Capítulo 5.

En este trabajo, se usó la herramienta WoPeD[2] [110] para comprobar si los servi-

[2]WoPed (del inglés *Workflow Petri Net Designer* es una herramienta de modelado para simular

Tabla 9.12: Verificación de las propiedades *inicio/fin*, *fuertemente conexa*, *segura* y *sin bloqueos* para procesos OWL-S con un nivel de composición: sustitución de la transición *run process* del patrón *proceso* por una HLPN que representa otro patrón. El signo '+' indica que se satisface la propiedad mientras que el '-' que no se verifica.

	inicio/fin	*fuertemente conexa*	*segura*	*sin bloqueos*
proceso	+	+	+	+
proceso + secuencia	+	+	+	+
proceso + separación	-	-	+	+
proceso + elección	+	+	+	+
proceso + separación-unión	+	+	+	+
proceso + sin-orden	+	+	+	+
proceso + si-entonces-sino	+	+	+	+
proceso + repetir-mientras	+	+	+	+
proceso + repetir-hasta	+	+	+	+

cios OWL-S verifican las propiedades anteriormente mencionadas y si, por lo tanto, cumplen su condición de WFs. Por ejemplo, la Figura 9.14 muestra la verificación de la coreografía de un proceso con una estructura de control del tipo *repetir-mientras*. Las tablas 9.12 y 9.13 muestran las propiedades que los servicio con un nivel de composición. Como resultado de este análisis se ha llegado a la conclusión de que la gran mayoría de estos procesos verifican las características que debe tener un flujo de trabajo. Sin embargo tres de ellos no cumplen con alguna de las siguientes propiedades:

- EL patrón *sin-orden* crea WFs que no están *bien estructurados* y no son *libres de conflictos*. Este patrón no separa correctamente el conflicto del paralelismo. Por lo tanto, al aplicar este patrón es necesario tener cuidado de no generar redes no deterministas. En las peores condiciones, la ejecución de un servicio con estas características podría dar lugar a resultados indeterminados.

- El patrón *separación* genera redes *inconexas*. La redes paralelas lanzadas por este patrón no están conectadas a la salida del flujo de trabajo. Por lo tanto, añade plazas de salida al flujo de trabajo. El problema está directamente relacionado con la especificación OWL-S que no define por completo el comportamiento de este patrón. La especificación OWL-S únicamente requiere lanzar la ejecución de las ramas paralelas pero no define ningún comportamiento ni restricción a dichas ramas de ejecución. De hecho, un servicio con una separación podría finalizar antes de que los procesos concurrentes hayan finalizado su ejecución. En opinión de este autor, el uso de patrones *separación* sin sincronización está desaconsejado.

Los resultados de las tablas 9.12 y 9.13 son aplicables a más niveles de composición: la composición actúa como una conjunción lógica. Por lo tanto, si un servicio tiene una secuencia de construcciones de control del tipo *repetir-mientras* y *elección*, la

y analizar el flujo de control de procesos. El sitio web oficial se encuentra en la siguiente dirección: http://www.woped.org/

Tabla 9.13: Verificación de las propiedades *viva*, *bien estructurada* y *libre de conflictos* para procesos OWL-S con un nivel de composición: sustitución de la transición *run process* del patrón *proceso* por una HLPN que representa otro patrón. El signo '+' indica que se satisface la propiedad mientras que el '-' que no se verifica.

	viva	*bien estructurada*	*libre de conflictos*
proceso	+	+	+
proceso + secuencia	+	+	+
proceso + separación	-	-	+
proceso + elección	+	+	+
proceso + separación-unión	+	+	+
proceso + sin-orden	+	-	-
proceso + si-entonces-sino	+	+	+
proceso + repetir-mientras	+	+	+
proceso + repetir-hasta	+	+	+

propiedad de *bien estructurada* no se verifica, ya que la conjunción de esta propiedad para los tres patrones es falso para uno de los casos.

9.4. Motor de orquestación de procesos OWL-S

La Figura 9.15 representa la arquitectura que da soporte a la ejecución servicios OWL-S. El objetivo de este motor es permitir tanto la gestión como la ejecución de la coreografía de procesos OWL-S. Es decir, la carga de servicios OWL-S y la ejecución de las redes de Petri que modelan su coreografía. Este último punto ha propiciado que este motor se haya construido a partir del motor OPENET4WF (ver Capítulo 6) al que extiende horizontalmente con la ontología OWL-S [277, 278]. De forma más detallada, los principales elementos de la pila de OWL-S son los siguientes:

- *Esquema*. Almacena el conjunto de clases, relaciones, propiedades y axiomas que describen los servicios OWL-S. Captura en F-Logic a la ontología OWL-S descrita en OWL.

- *Datos*. Este módulo almacena las instancias de los servicios descritos en OWL-S. Por ejemplo, almacena los procesos atómicos y compuestos definidos por un servicio.

- *Reglas*. Permiten *(i)* crear, modificar y borrar servicios OWL-S y *(ii)* crear la red jerárquica que captura la semántica operacional de la coreografía de procesos OWL-S. Al igual que en el caso de las redes jerárquicas con las HLPN, cada servicio OWL-S establece una relación con la red jerárquica que captura la coreografía del servicio. En este caso, el mapeado entre ambas pilas asocia una red de Petri plana (página) a cada uno de los patrones definidos para la semántica operacional de las estructuras de control de OWL-S y define las sustituciones para componer la red jerárquica.

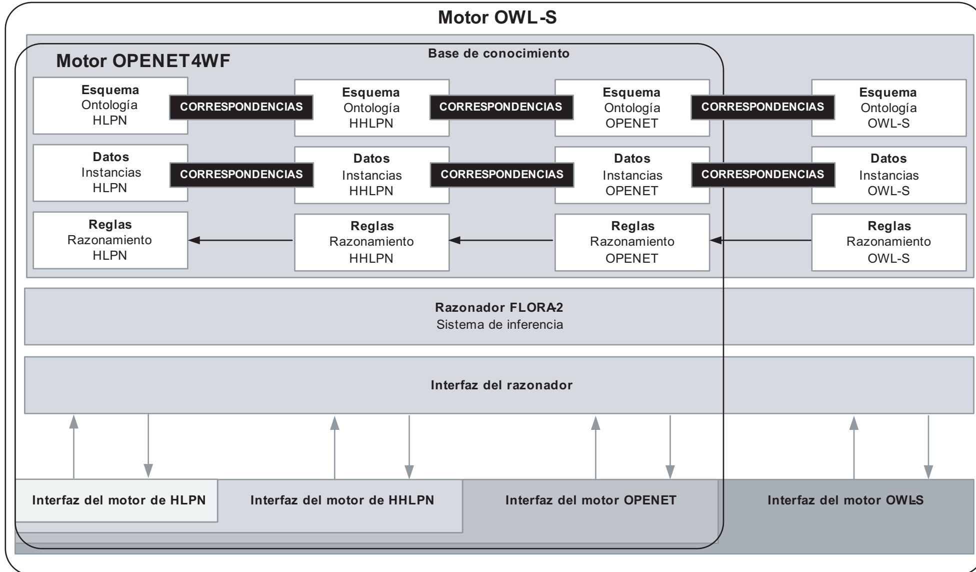


Figura 9.15: Arquitectura del motor de coreografías OWL-S. Este motor está construido sobre el motor OPENET4WF al que le hemos añadido otra capa para el manejo de OWL-S.

- *Interfaz del motor de OWL-S.* Esta fachada permite la carga y ejecución de servicios OWL-S desde un cliente. Para lograr tal funcionalidad, integra tanto a a la fachada Java de redes de Petri jerárquicas como planas. Ya que la semántica operacional está definida para redes planas, esta fachada primero usa la interfaz de redes jerárquicas para unir el conjunto de redes que definen la estructura del servicio y luego aplana la red jerárquica a una HLPN. La ejecución del servicio hace uso del gestor de ejecución definido en la arquitectura del motor de redes de Petri. Por ejemplo, a través de esta fachada es posible cargar la descripción del servicio OWL-S en formato OWL (Figura 9.16) y trasladar dicho servicio al modelo interno Java y posteriormente a través de la interfaz del razonador a F-Logic. En este ejemplo, esta interfaz se encargaría de crear los objetos del modelo que capturan el servicio de autenticación así como de definir sus entradas, salidas, precondiciones, etc.

9.5. Conclusiones

En este capítulo hemos presentado la adaptación del metamodelo de WFs propuesto en esta tesis doctoral para dar soporte a la ejecución de servicios OWL-S [277]. Desde la perspectiva de OWL-S la adaptación al metamodelo presenta grandes ventajas:

```
<service:Service rdf:ID="LogInService">
   <service:presents rdf:resource="#LogInProfile"/>
   <service:describedBy rdf:resource="#LogInProcess"/>
   <service:supports rdf:resource="#LogInGrounding"/>
</service:Service>

<process:AtomicProcess rdf:ID="LogIn">
   <process:hasInput rdf:resource="#Username"/>
   <process:hasInput rdf:resource="#Password"/>
   <process:hasOutput rdf:resource="#Ack"/>
   ...
</process:AtomicProcess>

<process:Input rdf:ID="Username">
   <process:parameterType rdf:datatype="&xsd;#anyURI">
      &xsd;#string
   </process:parameterType>
   <rdfs:label>Username</rdfs:label>
</process:Input>

<process:Input rdf:ID="Password">
   <process:parameterType rdf:datatype="&xsd;#anyURI">
      &xsd;#string
   </process:parameterType>
   <rdfs:label>Password</rdfs:label>
</process:Input>

<process:Output rdf:ID="Ack">
   <process:parameterType rdf:datatype="&xsd;#anyURI">
      &concepts;#Acknowledgement
   </process:parameterType>
   <rdfs:label>Ack</rdfs:label>
</process:Output>
```

Figura 9.16: Definición de un servicio de autenticación en OWL-S. El servicio descrito es atómico, toma por entrada un nombre de usuario y una clave, y devuelve un mensaje de confirmación.

- *Arquitectura.* Nuestro metamodelo proporciona una nueva arquitectura para la creación de servicios web semánticos en OWL-S. Ésta es una de las principales carencias de la especificación OWL-S, que indica cómo crear servicios y cómo coordinar sus procesos, pero no define una infraestructura que facilite la reutilización de partes de un servicio o la composición de distintos servicios OWL-S. Nuestra solución permite tratar los servicios como si fuesen WFs estructurando así el conocimiento que manejan a partir de las tareas (perfil del servicio) que van a realizar, los métodos (procesos) que las resuelven, su modelo de control (coreografía), sus dominios de aplicación y los recursos (operaciones del servicio) encargados de su ejecución. A través de nuestro metamodelo se posibilita la creación de una *librería* de componentes de servicio de forma que una coreo-

grafía, perfil o recursos pueda reutilizarse en distintos servicios a través de los mecanismos de adaptación existentes entre componentes.

- *Formalización.* La coreografía de procesos del modelo de servicio de OWL-S ha sido formalizada mediante HLPNs [269, 163]. Con esta formalización no existe ambigüedad acerca del comportamiento de las estructuras de control propuestas en OWL-S y, además, se han corregido algunas inconsistencias del modelo de servicio de OWL-S. Por ejemplo, OWL-S no indica cómo tratar las invocaciones que no cumplen con las precondiciones del servicio, ni tampoco define el comportamiento de la ejecución cuando la salida del servicio no encaja con ninguno de los efectos que hay que realizar en el entorno. El modelo propuesto soluciona los puntos anteriores dotando a los patrones denominados *proceso* y *resultado* de ramas alternativas para tratar estas situaciones.

- *Propiedades del modelo.* Hemos analizado el modelo de coreografía de procesos de OWL-S formalizado mediante redes de Petri y comprobado que la mayoría de los WFs construidos cumplen con las propiedades deseables, es decir, tienen una única plaza de entrada y de salida, son fuertemente conexas, seguras, sin bloqueos, vivas, bien estructuradas y libres de conflicto. Sólo dos de los patrones de comportamiento han demostrado no cumplir estas propiedades: el patrón *separación* de OWL-S y el patrón *sin-orden*. Por ello, se aconseja minimizar el uso de estos dos patrones en la medida de lo posible.

- *Ejecución.* El servicio se ejecuta como un WF de acuerdo con el modelo de ejecución visto en el Capítulo 6. En este sentido, tanto la estructura como el dinamismo del WF están soportados por el formalismo de las redes de Petri. En la bibliografía no existen muchos motores de OWL-S [199, 249] y ninguno de ellos basa su semántica operacional en un modelo formal: estos desarrollos simplemente proporcionan un API para cargar servicios OWL-S y ejecutarlos ocultando el detalle del comportamiento de algunas construcciones de control al programador. Este problema no existe en el metamodelo de WFs propuestos, ya que toda la semántica operacional (a excepción de las operaciones de los servicios) es explícita.

Desde otra perspectiva, esta aplicación nos permite apreciar la flexibilidad del metamodelo y del motor OPENET4WF para:

- *Extender la capa de patrones de WF con nuevos comportamientos.* Para añadir nuevos patrones de WFs es suficiente con crear una red de Petri que extienda el concepto `WFPattern` y el conjunto de axiomas que aseguren su estructura y comportamiento. En esta aplicación se han creado *cuatro nuevos comportamientos* definidos en OWL-S y no soportados por los patrones de WF de control básicos y avanzados. No obstante, alguno de ellos simplemente restringe una característica de un patrón básico. Por ejemplo, el patrón *separación* de OWL-S extiende el patrón *separación original* para identificar la plaza de salida y para indicar que no se requiere la finalización de las ramas concurrentes. Lo mismo

sucede con el patrón *elección* que mezcla la salida de las ramas concurrentes. Otros comportamientos requieren combinar varios patrones de WFs, como en el caso de los patrones *si-entonces-sino* o *sin-orden*, que se construyen combinando patrones como la *separación*, *elección*, *sincronización* o *mezcla*.

- *Extender el motor* OPENET4WF *para transformar servicios OWL-S en WFs.* Como hemos visto en esta aplicación, el motor OPENET4WF se puede extender con facilidad para dar soporte a otras especificaciones y transformarlas en WFs. Para ello es suficiente con extender la base de conocimiento con el conjunto de predicados, axiomas y reglas que permitan cubrir el nuevo modelo y su transformación a WF. En el caso de OWL-S se añadió una capa para el manejo de la ontología OWL-S y las reglas que permiten pasar los perfiles del servicio, modelos de servicio y modelo de conexión a las tareas, métodos, modelos de control y recursos. Asimismo, la capa Java facilita el paso de OWL-S y las ontologías del dominio que anotan el servicio al modelo de representación del conocimiento de FLORA-2.

Conclusiones

A continuación analizaremos la consecución de los objetivos planteados para nuestra investigación, descritos en el prefacio de esta memoria, y las conclusiones acerca de los resultados obtenidos.

El primer objetivo que se planteó fue dotar OPENET4WF con una arquitectura conceptual para dar soporte al modelado de flujos de trabajo (WFs) que facilitase la reutilización del conocimiento y que permitiese razonar semánticamente acerca de las características de los WFs. Este objetivo se ha alcanzando adaptando la arquitectura de UPML [104], pensada para el modelado de sistemas basados en conocimiento (SBCs), a las características diferenciadoras de los WFs. Para ello fue necesario extender UPML con dos nuevas dimensiones [271] para modelar adecuadamente *(i)* la coordinación entre las distintas tareas que componen el WF y *(ii)* la coordinación entre los agentes y las tareas a realizar. Nuestro metamodelo describe un WF mediante de cinco componentes de conocimiento: tareas, métodos, modelos de control, modelos de recursos y modelos del dominio. Cada uno de estos componentes captura una de las características de los WFs: las tareas describen su *funcionalidad*, los métodos y modelos de control definen su *comportamiento*, los modelos de recursos proporcionan la *organización* de los recursos que participan en la ejecución del WF, y finalmente los modelos del dominio aportan el *conocimiento* del dominio de aplicación.

Nuestro metamodelo también proporciona altos índices de reutilización. Esta capacidad se consigue a través de dos propiedades heredadas de la arquitectura UPML:

- *Independencia.* Se garantiza que cada componente de conocimiento es autónomo, es decir, su definición es independiente de los demás componentes del metamodelo. Esta propiedad posibilita la existencia de librerías de componentes reutilizables a partir de las que un diseñador pueda seleccionar, por ejemplo, el modelo de control más adecuado para modelar la coordinación del conjunto de (sub)tareas de un método.

- *Adaptación.* Las relaciones entre los distintos componentes de un WF se establecen a través de los *adaptadores*. Un adaptador asocia dos componentes de conocimiento, y puede ser de dos tipos en función de su finalidad. Por una parte, los *refinadores* conectan componentes de conocimiento del mismo tipo

y permiten crear un nuevo componente extendiendo la definición de uno ya existente. Con este mecanismo se reduce el tiempo de desarrollo de nuevos componentes de conocimiento, ya que se puede partir de uno ya creado. Por otra parte, los *puentes* proporcionan el mecanismo para combinar dos componentes de conocimiento de distinto tipo. A través de estos puentes, se unifica la terminología empleada en los dos componentes, de forma que puedan "entenderse". Por ejemplo, el puente entre un método y un modelo de control permite asociar las entradas, salidas, competencia y (sub)tareas del método con los elementos correspondientes en la firma de la red de Petri que modela el control.

Las ontologías [122] proporcionan la capacidad semántica del metamodelo. Cuando se define un componente de conocimiento, la terminología utilizada en su descripción pertenece a una determinada ontología. Esta característica facilita la labor de los adaptadores, ya que normalmente se limitan a establecer las correspondencias entre los conceptos que anotan los distintos componentes. Además, otra de las ventajas de nuestro metamodelo es que ha sido definido como una ontología para el modelado de WFs. Así, es posible razonar acerca de sus características semánticas. Por ejemplo, comparar semánticamente las funcionalidades definidas por dos tareas.

Como segundo objetivo, nos planteamos integrar dentro del metamodelo un modelo de representación de WFs con alta capacidad expresiva y legibilidad para así facilitar la definición de las estructuras de control complejas (las cuales no se pueden modelar adecuadamente con los mecanismos tradicionales de modelado de SBC). Este objetivo se consiguió utilizando las redes de Petri [190] como formalismo de representación de WFs. Como señalamos en el Capítulo 2, las redes de Petri, y en particular las de alto nivel [144] y jerárquicas [149], tienen una alta capacidad expresiva y de legibilidad, al disponer de un modelo de concurrencia formal y visual. Además, nos proporcionan mecanismos de análisis a través de los cuales es posible conocer las propiedades de la red, las dependencias entre las transiciones o, por ejemplo, si es posible alcanzar un determinado estado.

Aunque las redes de Petri son un marco adecuado para el modelado de WFs, su integración en el metamodelo no ha sido directa, ya que no están descritas a través de un formalismo lógico, y por lo tanto, una máquina no puede razonar con ellas ni procesarlas directamente. Para solucionar este problema se creó una ontología de redes de Petri de alto nivel (HLPNs) y otra de HLPNs jerárquicas [266]. Además, hemos fundamentado la ontología en el estándar ISO/IEC 15909-1 [144]. Además, es compatible con el estándar de intercambio ISO/IEC 15909-2 [145] lo cual garantiza que pueda utilizarse en las herramientas de diseño de redes de Petri disponibles en el mercado.

Los conceptos definidos en estas ontologías permiten representar la estructura de la red y también su dinámica de ejecución. Así, es posible analizar y simular el funcionamiento de una red, y localizar comportamientos no deseados en un WF. Para garantizar que tanto la estructura de la red como su ejecución son correctas, la ontología dispone de un conjunto de axiomas. En este sentido, se puede afirmar que

ésta es una ontología pesada, y por ello, no todos los razonadores disponibles en el mercado son capaces de manejarla. Por ejemplo, los razonadores basados en OWL no permiten representar muchos de los axiomas de la ontología de HLPNs.

Como tercer objetivo de esta tesis doctoral se planteó la creación de una ontología que facilitase la definición y composición de WFs a partir de patrones o modelos de comportamiento conocidos y aceptados por los usuarios. Este objetivo se consiguió añadiendo una capa de patrones de comportamiento por encima de la ontología de HLPNs jerárquicas. Como ya se comentó, el modelo de control de nuestro metamodelo está basado en HLPNs jerárquicas y, aunque estas redes son muy expresivas y tienen una gran legibilidad, consideramos necesario definir una capa en la cual se incluyesen los principales patrones de comportamiento usados para modelar WFs. De esta forma el diseño se simplifica, ya que permite crear los WFs a partir de un mayor nivel de abstracción, es decir, los patrones. Al igual que un lenguaje de programación estructurada sin estructuras de control del tipo *secuencia*, *si-entonces-sino*, *repetir-mientras*, o *for* es poco más que un lenguaje ensamblador, los patrones aportan ese nivel de abstracción añadido a las HLPNs. En esta situación, un diseñador puede construir su modelo a partir de piezas independientes como si se tratase de un rompecabezas.

El cuarto objetivo fue proporcionar un modelo de ejecución sin ambigüedades que se construya a partir de la combinación los distintos (e independientes) componentes de conocimiento del metamodelo, e implementar un motor de ejecución de WFs que interprete dicho modelo. El plantear la coordinación entre las tareas de un WF en términos de redes de Petri ha facilitado la consecución de este objetivo. El metamodelo de WFs descrito en esta memoria permite crear un modelo de ejecución a partir de la integración de las distintas componentes de conocimiento [270], las cuales se integran a través de los puentes existentes en el metamodelo, dado como resultado la HLPN jerárquica que representa el modelo de ejecución.

Si partimos de una tarea o funcionalidad a resolver, la composición del WF consiste en *(i)* seleccionar el método que ejecutará la tarea, *(ii)* el modelo de control que coordinará las (sub)tareas del método, *(iii)* el modelo del dominio en el que se aplica el WF y *(iv)* el modelo de recursos que contiene los agentes encargados de su ejecución. Claro está que el núcleo del modelo de ejecución serán las distintas páginas del modelo de control que se asociarán con las entradas, salidas, pre y postcondiciones del método y con los criterios de asignación de recursos aplicables a cada tarea.

Al ser el modelo de ejecución una HLPN jerárquica, el núcleo de la infraestructura tecnológica para la ejecución de OPENET4WF es un motor de redes de HLPNs construido sobre un razonador basado en lógica de marcos. Esta arquitectura software tiene la ventaja de ser fácilmente escalable, como se ha podido ver en el Capítulo 8: para añadir una nueva capa por encima del metamodelo de WFs es suficiente con crear su base de conocimiento, sus reglas y su interfaz de acceso.

Para facilitar la integración de los recursos que ejecutarán las tareas del WF, la última capa de la infraestructura se define a través de una arquitectura orientada a servicios. Cada recurso, independientemente de su tipo, se puede conectar al gestor de mensajes de OPENET4WF y obtener el estado del WF, las tareas de su cola de trabajo o, por ejemplo, comunicar al sistema la realización de un trabajo.

Como quinto y último objetivo se persigue validar nuestra propuesta mediante WFs con distintas características y en distintos dominios de aplicación. La validación del metamodelo ha sido una de las prioridades de esta tesis doctoral, y ha tenido por objetivo *(i)* verificar la aplicabilidad y generalidad de nuestra propuesta de modelado de WFs, y *(ii)* evaluar las distintas ontologías que componen el metamodelo. En el primer punto comprobamos que el metamodelo es lo suficientemente expresivo para representar los WFs clásicos que soportan la mayoría de los sistemas comerciales y, que además permite el modelado de WFs *ricos en conocimiento*. Esta validación se realizó a través de tres aplicaciones:

- Los desarrollos en el dominio de la industria del mueble permitieron comprobar la capacidad del metamodelo para representar conocimiento experto, típico de cualquier dominio dedicado a la fabricación. La representación de este conocimiento es de gran utilidad en la resolución de WFs complejos, y es uno de los rasgos que diferencian nuestro metamodelo respecto de los existentes en el mercado. Además, también nos permitió comprobar su capacidad de configuración y reutilización. Es importante resaltar esta propiedad, ya que en la actualidad la rapidez en la adaptación a los cambios es de suma importancia para las empresas. Por ello, disponer de un sistema que facilite una transición rápida y relativamente sencilla es un valor añadido.
- El modelado de unidades de aprendizaje en el dominio de la educación demostró la capacidad del metamodelo y de OPENET4WF para dar soporte a WFs complejos que pueden llegar a tener más de dos mil nodos. Como resultado de esta aplicación, creamos uno de los pocos sistemas que permiten la ejecución de unidades de aprendizaje basadas en el estándar IMS LD, y la única cuyo modelo de ejecución está formalizado.
- El modelado como WFs de los servicios web semánticas expresados en OWL-S ha permitido comprobar la facilidad con la que se puede extender la capa de patrones de WFs definida en nuestro metamodelo. Esta capacidad es indispensable, ya que simplifica la incorporación de nuevos comportamientos al metamodelo. Con este desarrollo también se demuestra la versatilidad del metamodelo y del motor que lo implementa, ya que nos permite considerar los servicios web como una (sub)categoría de WFs donde no hay intervención humana.

La validación de las ontologías de HLPNs, de redes jerárquicas y de patrones de comportamiento de WFs se realizó tomando como conjuntos de pruebas a ejemplos de las tres aplicaciones anteriormente mencionadas.

Trabajos futuros

Aunque OPENET4WF ha demostrado ser una herramienta de gran utilidad para el desarrollo y ejecución de WFs, hemos identificado una serie de aspectos sobre los que sería interesante investigar en qué medida la representación semántica de los WFs facilitaría su resolución. En otras palabras, analizar si el metamodelo propuesto ayuda a resolver los siguientes tópicos:

- Facilitar la selección automática *(i)* del método más adecuado para resolver una tarea, y *(ii)* del modelo de control que coordina su ejecución. Esta selección debería basarse tanto en la tarea como en las características del modelo del dominio al que se va aplicar y de los agentes que participarán en la ejecución.
- Utilizar técnicas de aprendizaje automático o minería de datos para mejorar la elección del agente más adecuado para la ejecución de un método primitivo. En este sentido, se trataría de reforzar la monitorización del usuario y la inferencia de conocimiento a través del estudio de su comportamiento.
- Automatizar operaciones como el descubrimiento o la composición de WFs. Este punto está muy relacionado con investigaciones ya existentes en el campo de los servicios web y las características propias de los WFs podrían enriquecer la resolución de estas operaciones. Además, sería un paso fundamental hacia la simplificación del proceso de desarrollo ya permitiría al usuario obtener un WF únicamente a partir de la tarea, del modelo del dominio y del modelo de organización.

A

Aprendizaje de tiempos

La estimación de los tiempos de procesado afecta a gran parte del ciclo de vida del diseño del producto [131]: diseño, planificación, producción, ensamblaje, etc. Por ejemplo, estos tiempos se tienen en cuenta a la hora de rediseñar el producto si el tiempo de fabricación es mayor que el previsto. Los tiempos de procesado se aproximan a partir de los diseños del producto, bien sea a través de *(i)* diseños CAD que permiten una planificación detallada y la simulación de algunos procesos de fabricación [182, 35, 36] o a través de *(ii)* diseños conceptuales que contienen la información crítica del diseño pero con una menor precisión [35, 36] debido a que un diseño detallado es muy caro de obtener tanto en coste como en tiempo. El material, la fabricación y los requerimientos de ensamblaje que se infieren a partir de los diseños afectan al tiempo de procesado. El largo, ancho, espesor y el resto de las entradas que definen las piezas de un mueble son algunas de las variables que tienen influencia directa en el tiempo de procesado de una operación. Estas variables, que se extraen del despiece del mueble, pueden además combinarse para dar lugar a nuevas variables como puede ser la superficie o el volumen. Por ejemplo, el tiempo de procesado para barnizar una pieza depende de la superficie de la misma. Si además esta pieza se barniza a mano entonces el tiempo de manipulación se ve afectado también por el volumen de la pieza. Por lo tanto, el tiempo de procesado de una máquina puede depender de varias variables de entrada, como las dimensiones del producto, el material, la velocidad de la máquina para procesar cada tipo de material, etc. De este modo, una máquina puede tener varias funciones de regresión (una para cada tipo de producto) para la estimación del tiempo de procesado. Por lo tanto, una buena estimación del tiempo de procesado de una máquina requiere una función de regresión precisa y además, la selección de la función más apropiada de entre todas las funciones de regresión de la máquina.

La aproximación manual a la estimación de estos tiempos descrita en el Capítulo 7 resultó ser sumamente costosa, ya que implicaba *(i)* mediciones temporales a pie de máquina y *(ii)* la identificación de los parámetros que influyen en dichas operaciones. En este contexto, la no identificación de una de estas variables repercutía muy negativamente en la calidad de la fórmula, la cual aplicada a la fabricación a gran escala, derivaba en un error en la estimación. Por ejemplo, las tablas A.1 y A.2 muestran el error medio de esta aproximación para centros de coste y máquinas respectivamente.

Tabla A.1: Error medio por centro de coste

Centro de coste	**Porcentaje de error**
Seccionadora	+16,1
Canteadora	-5,8
Centro de Mecanizado	+0,20
Lijadora Calibradora	-14,80
Lijadora de Cantos	-15,8
Maquinas de Madera	-2,3
Ebanistería	+28,6
Acabado	+8,46

Tabla A.2: Error medio por máquina

Máquina	**Porcentaje de error**
Seleccion y servir madera	-5,0
Seleccion y servir tablero	-5,0
Seccionadora	-8,0
Seccionadora de madera	-22,0
Plana	+11,4
Escuadradora	+3,4
Canteadora	-14,5
Lijadora Calibradora	-5,5
Lijadora de Cantos	-33,5
Taladro Pequeño	-18,2
Espigadora	-40,2
Tupí	+5,6
Centro de Mecanizado	+7,6
Montaje Intermedio	-26,9
Fondo Manual	-11,2
Acabado Manual	-15,8
Prensa Hidráulica de Cascos	+10,6
Montaje Final	+31,0
Embalaje retráctil	-16,0
Embalaje manual	-22,5

Los valores positivos indican una sobre-estimación del tiempo de procesado mientras que los negativos indican que se subestima el tiempo. En este tipo de estimaciones un error medio de menos del 5 % es asumible, pero como puede apreciarse en dichas tablas, estos errores estaban lejos de ser óptimos y podían suponer la diferencia entre obtener beneficios o sufrir pérdidas. Además, la generación de una base de conocimiento y su continua adaptación no era asumible, ya que requería un esfuerzo que los expertos de la empresa no podían asumir.

Dada las limitaciones de la aproximación anterior, se buscó reducir el error de estas estimaciones a través de la mejora continua de las fórmulas. Para ello se partió de las fórmulas manuales definidas por los expertos y se procedió a su desarrollo a través de un proceso de aprendizaje automático. En este capítulo describimos la aproximación que permite aprender de forma automática las fórmulas empleadas para la estimación de tiempos, mejorar la precisión de las estimaciones y, en cualquier caso, mantener la estructura de conocimiento propuesta por los expertos de forma que sus resultados sean fácilmente interpretables [189, 188].

A.1. Estimación de tiempos de procesado en la industria del mueble

La estimación de los tiempos de procesado es de gran importancia en la industria del mueble. En esta industria, estos tiempos son más difíciles de calcular, ya que influyen muchos factores difícilmente medibles; al contrario de otras industrias, estas estimaciones no pueden calcularse únicamente a través de herramientas de simulación, ya que la componente humana tiene demasiada influencia en este tipo de fabricación. Por ello, estas estimaciones se realizan a partir del conocimiento experto de fabricación.

Las estimaciones del tiempo de procesado suelen utilizarse en múltiples puntos del ciclo de vida de un mueble, sin embargo, tienen una especial influencia en la fase de planificación. Muchas de las decisiones que se toman a partir de un plano de trabajo se infieren a partir de las estimaciones de tiempo de procesado de cada uno de los procesos de fabricación. Por ejemplo, la información de un plano puede utilizarse durante el rediseño de un mueble porque el tiempo de procesado de alguna de sus piezas resulta demasiado costoso. Por lo tanto, si se quiere automatizar su cálculo, es fundamental que un experto pueda extraer fácilmente la información acerca de las distintas estimaciones de tiempo de procesado que están implícitas en el plano. La simplicidad de la extracción de información a partir de una función de regresión depende de la representación del conocimiento contenido en dicha función. En el caso de la fabricación de muebles, los expertos demandan funciones de regresión en las cuales sea fácil evaluar el peso o contribución de cada una de las variables en el tiempo de procesado. Es más, el descubrimiento de cómo las distintas variables que afectan al tiempo de procesado de una máquina se interrelacionan es de gran utilidad para el experto. En este sentido, también es necesario tener en cuenta que los expertos estructuran su conocimiento para la estimación de tiempos en polinomios construidos a partir de las distintas variables de entrada de cada una de las máquinas. Por lo tanto, una representación similar es clave para la interpretación del tiempo resultante. Resumiendo, la solución para la estimación de tiempos de procesado del sistema construido debía tener en cuenta dos premisas principales: gran precisión pero también facilidad a la hora de extraer e interpretar la información.

Para clarificar el procedimiento usado para realizar el cálculo del tiempo de procesado de una operación, se hará uso de la mesa cuyo diseño conceptual está representado en la Figura A.1. Mencionar que la estimación del tiempo total de fabricación del mueble necesita primero definir las operaciones a realizar para posteriormente seleccionar la ruta que establecerá el orden de cada una de estas operaciones. Para la mesa ejemplo, el tiempo de fabricación vendrá dado de forma resumida por los tiempos de procesamiento de las siguientes operaciones:

- Cortar las piezas a partir de tableros de madera.
- Aplanar y calibrar su superficie y espesor.
- Cortar las uniones de las piezas en ángulos de 45 grados.
- Cortar una ranura en cada una de las piezas a unir.

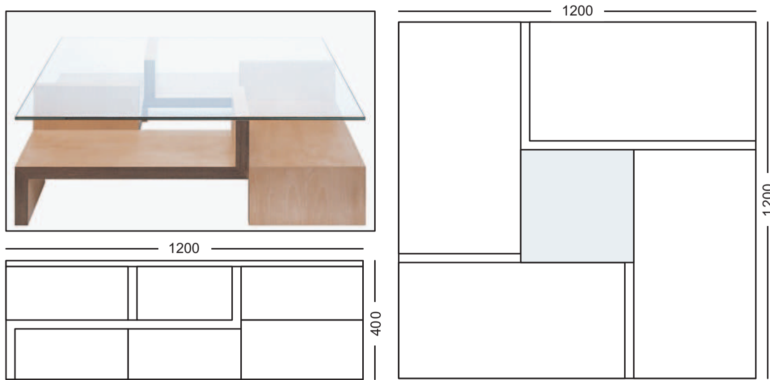


Figura A.1: Diseño conceptual a partir del cual se deducen las partes del mueble y los procesos de fabricación

- Realizar el acabado.
- Sujetar la unión con un adhesivo fuerte.
- Ensamblar y empaquetar el mueble.

El tiempo de procesado de cada una de estas etapas está influenciado por la operación a realizar y por el recurso o máquina que va a realizar dicha operación. Por ejemplo, las piezas de madera de la mesa a fabricar deben calibrarse para ajustarse al espesor y precisión requerida por los diseños. La Figura A.2 representa la *máquina lijadora calibradora* que se encarga de realizar dicha operación. Este tipo de máquina tiene como características una velocidad de avance, una velocidad de abrasión, un tamaño de cinta transportadora, un tamaño de espesor máximo de trabajo, y unas dimensiones máximas y mínimas de las piezas. Sin embargo, en este caso únicamente la velocidad de avance tiene influencia en el tiempo de procesado, ya que los otros parámetros únicamente discriminan el tipo de recurso que puede realizar la operación. En condiciones perfectas, el tiempo de procesado podría calcularse directamente multiplicando la velocidad de avance por la longitud de las piezas a calibrar. Sin embargo, la manipulación de las piezas reduce considerablemente la velocidad de procesado. En este ejemplo, aunque la velocidad de avance de la máquina es de 5 metros por minuto, los tiempos medidos en fábrica mostraron que dicha velocidad dependía directamente de la superficie de las piezas.

A.2. Polinomios de estimación del tiempo de procesado

La aproximación a la estimación de tiempos de procesado que se presenta a continuación tiene por objetivo la definición de un modelo de regresión lineal de gran precisión pero que a su vez mantenga la estructura del conocimiento proporcionado

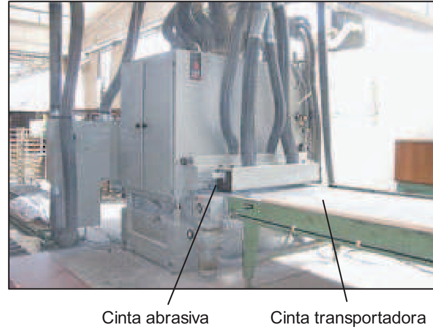


Figura A.2: Máquina lijadora calibradora

por el experto. El objetivo de esta última afirmación es facilitar a los expertos tanto la comprensión como la extracción de información a partir de las fórmulas de regresión. La información aportada por estas estimaciones permite a los expertos implementar o modificar un determinado plan de producción y por ello la simplicidad de estas funciones de regresión es tan importante como la precisión de las mismas. Para facilitar dicho objetivo, las funciones deben:

- Asociar un peso o contribución a cada variable de entrada usada para la estimación.
- Permitir la generación de nuevas variables generadas a partir de las variables de entrada. Por ejemplo, el producto de las tres dimensiones de una pieza genera la variable *volumen*

Por ello, los expertos estructuran su conocimiento para estimar los tiempos de procesado por medio de polinomios del estilo:

$$\textit{Tiempo de procesado} = 100 \cdot \textit{largo} + 200 \cdot \textit{volumen} \tag{A.1}$$

Además, desde la perspectiva del experto, es muy importante mantener esta representación del conocimiento de cara a poder extraer la información de las funciones de regresión. Por ejemplo, en función de cierto valor de una variable, el experto puede decidir realizar una operación en una determinada máquina. Siguiendo estos pre-requisitos, los tiempos de procesado de una máquina pueden describirse como polinomios con varias variables de entrada que pueden combinarse de múltiples formas:

$$\sum_i \alpha_i \cdot \prod_{j=1}^{na} x_j^{\delta_{i,\,j}} \tag{A.2}$$

donde α_i son los coeficientes, x_j, $j = 1, \ldots, na$, son las variables de entrada, y $\delta_{i,\, j}$ es un indicador variable definido por:

$$\delta_{i,\, j} = \begin{cases} 1 & \textit{if } x_j \in \textit{i-ésimo término del polinomio} \\ 0 & \textit{en otro caso} \end{cases} \tag{A.3}$$

Es más, para una determinada máquina, pueden coexistir diferentes polinomios, cada uno de los cuales representa la estimación del tiempo de procesado de una clase de variables de entrada. Por ejemplo, para una determinada máquina, los tiempos de procesado de una pieza con un espesor por encima de un umbral determinado se estiman mediante un polinomio, mientras que por debajo de ese umbral lo hacen con uno distinto. Por ello, el proceso de aprendizaje debe de obtener una función de regresión para cada clase de variables de entrada de una máquina.

A.3. Modelo de reglas TSK para la estimación de tiempos de procesado

A partir de una entrada, el sistema debe seleccionar la función de regresión de la máquina y luego estimar su tiempo de procesado. Dado que la función de regresión debe mantener la estructura definida en la Ecuación A.2, el modelo más conveniente para aproximar estas estimaciones es el modelo de reglas borrosas de Takagi-Sugeno-Kang (TSK) [244, 243]. En una regla TSK, la salida (consecuente) es una función construida a partir las variables de entrada (antecedente). Por lo tanto, este modelo proporciona los mecanismos necesarios para estimar los tiempos pero manteniendo la estructura del conocimiento proporcionado por el experto. El modelo de reglas propuesto es el siguiente:

$$\begin{aligned} R_k :\ & \text{SI } X_1\ es\, A^1_{l^k_1} \text{ Y } \ldots \text{ Y } X_{na}\ es\, A^{na}_{l^k_{na}} \\ & \text{ENTONCES } Y\ es\ \sum_i \alpha_i \cdot f_i\left(x_1, \ldots, x_{na}\right) \end{aligned} \tag{A.4}$$

donde X_j es una variable lingüística, $A^j_{l^k_j}$ es una etiqueta lingüística de esa variable, $l^k_j = 1, \ldots, nl_j$, Y es una variable de salida, y $f_i\left(x_1, \ldots, x_{na}\right)$ son funciones de las variables de entrada (x_j).

Estas funciones, f_i, pueden definirse de dos formas:

$$f_i\left(x_1, \ldots, x_{na}\right) = \begin{cases} \sum_{j=1}^{na} x_j \cdot \delta_{i,\, j} \\ \prod_{j=1}^{na} x_j^{\delta_{i,\, j}} \end{cases} \tag{A.5}$$

permitiendo todas las posibles combinaciones necesarias para estimar los tiempos de procesado.

La generación de la base de conocimiento de reglas TSK se describe en la ecuación A.4 y requiere la definición de varias etiquetas lingüísticas, coeficientes (α_i), y también funciones (f_i) con muchas estructuras muy diferentes. El uso de conocimiento experto ayuda a la reducción del espacio de búsqueda, por ejemplo limitando las estructuras válidas para las funciones.

A.3.1. Aprendizaje de bases de conocimiento TSK

En este trabajo, se ha optado por realizar el aprendizaje a través de un algoritmo evolutivo. El aprendizaje de bases de conocimiento borrosas mediante algoritmos evolutivos ha demostrado ser una técnica poderosa [77]. En este campo, los algoritmos evolutivos tienen ventajas respecto a otros métodos de aprendizaje: primero, las reglas de la base de conocimiento se pueden representar de diferentes formas, debido a la flexibilidad en la representación de las soluciones; por otro lado, otra significativa ventaja es que permiten balancear la simplicidad de extracción de la información con la eficiencia a la hora de aprender reglas a través de distintos algoritmos. De acuerdo a [77], existen varias aproximaciones a la hora de representar la solución de un problema en el aprendizaje evolutivo de bases de conocimiento: Pittsburgh, Michigan, Iterative Rule Learning (IRL), y cooperativa-competitiva. En la aproximación Pittsburgh [53], cada cromosoma codifica una base de conocimiento completa. La longitud de los cromosomas puede ser variable, ya que permite representar bases de reglas con un número variable de reglas. Esta metodología tiene un importante coste computacional, ya que en cada iteración se tienen que evaluar muchas bases de conocimiento. En la aproximación Michigan [157], las reglas pueden evolucionar a lo largo del tiempo debido a su interacción con el entorno utilizando algoritmos evolutivos y aprendizaje con refuerzo. Sin embargo, la distribución de la recompensa entre los individuos es normalmente compleja. En cambio, en la aproximación IRL [76], no existe redistribución de la recompensa: el algoritmo evolutivo aprende una única regla y no toda la base de reglas. Después de cada secuencia de iteraciones, se selecciona la mejor regla y se añade a la base de reglas final. La regla seleccionada ha de ser penalizada para inducir la formación de nichos en el espacio de búsqueda. Finalmente, en la aproximación cooperativa-competitiva [117], las reglas evolucionan juntas (de forma cooperativa al contrario de la aproximación IRL), pero compiten entre ellas con el fin de obtener mayor bondad. En esta aproximación, es imprescindible incluir un mecanismo para mantener la diversidad de la población (inducción de nichos). Este mecanismo debe garantizar que existe competición entre los individuos del mismo nicho, pero también debe evitar la eliminación de los individuos más débiles que ocupan un nicho que no está cubierto por otros individuos de la población.

Existen diferentes propuestas en la bibliografía para el aprendizaje de bases de conocimiento TSK mediante algoritmos evolutivos: en [75] se presenta un proceso evolutivo en dos fases para el diseño de systemas de reglas borrosas TSK a partir de ejemplos. Combinan una fase de generación basada en una estrategia evolutiva (μ; λ), y una fase de refinamiento en la cual se adaptan mediante un proceso evolutivo híbrido tanto a los antecedentes como a los consecuentes de las reglas de la base de

conocimiento. En [133] se utiliza una aproximación basada en programación genética para la identificación de estructuras en modelos neurales borrosos mediante reglas TSK. El objetivo es la obtención de una partición óptima del espacio de entrada dentro de conjuntos borrosos ortogonales y gausianos. En [200] se describe un algoritmo genético para la generación de modelos TSK construidos en tres fases: aprendizaje de la estructura, simplificación de la base de reglas y ajuste fino. En[309] se propone otra aproximación con un algoritmo que integra el aprendizaje local y global. Este algoritmo utiliza la idea de regresión pesada local y aproximación local en estadística no paramétrica. Este método es capaz de ajustar sus parámetros basándose en las preferencias del usuario y generar modelos que compensan el ajuste global y la interpretación local. Finalmente, en [171] se realiza el aprendizaje de controladores borrosos tipo TSK mediante una aproximación híbrida que consiste en un algoritmo de auto-organizado de segmentación (clustering) y un forma dinámica de evolución simbiótica. Primero, se identifica la estructura interna y posteriormente se le aplica un método evolutivo de búsqueda secuencial.

A.3.2. Adaptación del modelo TSK

En este trabajo, las estimaciones temporales se han aproximado mediante reglas TSK. Sin embargo, los consecuentes de este tipo de reglas son generalmente polinomios de primer orden mientras que en nuestro problema se necesitaba una mayor flexibilidad, ya que los términos de los polinomios pueden ser sumas o productos de variables. Por consiguiente, fue necesario adaptar el modelo de reglas TSK para este problema.

El diseño de reglas TSK de estructura variable es una tarea compleja, ya que es necesario crear etiquetas para la parte antecedente, para los coeficientes del polinomio, pero también para las funciones de las variables de entrada de cada uno de los términos del polinomio. Para aprender esta base de conocimiento, el algoritmo debe poseer la capacidad de representar reglas con diferente estructura, aspecto que proporciona la programación genética. La programación genética es un algoritmo evolutivo que representa cada cromosoma de la población como un árbol de longitud variable.

La flexibilidad en la estructura de los consecuentes de las reglas TSK es fundamental para la estimación de los tiempos de procesado, pero se imponen algunas restricciones a esta estructura, ya que no todas son válidas. En nuestro caso hemos usado una gramática libre de contexto para definir los polinomios encargados de estimar los tiempos de fabricación. Esta gramática restringe la expresividad de la fórmula y, por lo tanto, su espacio de búsqueda. En la actualidad, cada operación tiene una gramática diferente que contiene un subconjunto de las posibles variables de entrada. Las variables tenidas en cuenta para cada operación vienen dadas por el conocimiento de los expertos. Aunque de esta forma se está mejorando la eficacia computacional, también es cierto que posibilita que puedan perderse algunas variables relevantes. Sin embargo, los numerosos experimentos llevados a cabo hasta la fecha con datos de laboratorio han resultado totalmente satisfactorios. Si bien es complicado concluir si ello se debe *(i)* a que todas las variables de influencia del polinomio estaban ya identificadas o *(ii)* a que los coeficientes de las variables identificadas del polinomio

para cada una de las clases identificadas por el algoritmo reducen el efecto provocado por la falta de una variable.

A.4. Método para el aprendizaje de tiempos

Este método se encarga de aprender un conjunto de bases de reglas que permiten estimar el tiempo de procesado de una operación por parte de una determinada máquina. Cada una de las reglas tendrá una estructura TSK, pero la estructura del consecuente de las reglas puede ser muy diferente de una regla a otra. Aunque esta estructura tiene restricciones: por ejemplo, una función (f_i) no puede ser la combinación de sumas y productos (sólo de sumas o productos). Por esta razón, el algoritmo evolutivo más adecuado para evolucionar estas fórmulas es la programación genética [158]. En programación genética, un individuo es un árbol de longitud variable que puede tener una estructura diferente donde las restricciones acerca de la estructura del cromosoma se resuelven usando una gramática libre de contexto.

Para nuestro problema de aprendizaje que se resolvió en este trabajo, la aproximación Pittsburg resultaba demasiado costosa. Era más adecuado optar por una aproximación que codificase una única regla en cada individuo de la población. La aproximación seguida en fue la cooperativa-competitiva. También se optó por el mecanismo de *token competition* [303, 167] para mantener la diversidad. Conforme a [29], este mecanismo es adecuado para la programación genética, ya que para esta clase de algoritmos evolutivos la estructura de los individuos puede ser completamente diferente y, por lo tanto, la evaluación de las similaridades compleja. Otros mecanismos usados para mantener la diversidad, como *crowding* o *bondad compartida*, deben mantener la similaridad entre pares de individuos por lo que son difícilmente aplicables a este problema.

El aprendizaje se basa en un conjunto de ejemplos de entrenamiento. Cada uno de los ejemplos de este conjunto contiene un conjunto de *tokens*. En el mecanismo de *token competition* cada uno de los individuos que cubre un ejemplo debe competir por agarrar sus *tokens*, pero únicamente uno de ellos (el más fuerte) puede conseguirlo. Durante el proceso evolutivo, el individuo más fuerte de cada nicho intentará obtener el máximo número de *tokens* del nicho con el fin de incrementar su fuerza (y su bondad). Por el otro lado, el individuo más débil reducirá su fuerza al no poder competir con el mejor individuo del nicho.

Los ejemplos de entrenamiento se representan mediante tuplas: ($entrada_1$, ..., $entrada_n$, *salida*), donde los primeros n elementos representan las variables de entrada y el último la variable de salida. Las variables de entrada suelen ser distintas para cada una de las máquinas analizadas. Por este motivo, y para facilitar la comprensión de la solución, se tomarán en consideración únicamente tres variables de entrada: *largo*, *ancho* y *espesor* de una pieza.

A.4.1. Descripción de la gramática libre de contexto

En programación genética cada uno de los individuos es un árbol de tamaño variable y por ello la estructura de los individuos puede ser diferente. En este sentido, es necesario especificar un conjunto de restricciones que faciliten la generación de individuos válidos para la población y que aseguren la estructura de los individuos después de las operaciones de cruce y mutación. A través de una gramática libre de contexto, se pueden definir todas las estructuras válidas del árbol (cromosoma) de una forma compacta. Una gramática libre de contexto es una cuádrupla (V, Σ, P, S), donde V es un conjunto finito de variables, Σ es un conjunto finito de símbolos terminales, P es un conjunto finito de reglas de producción y S es un elemento de V llamada variable de inicio.

La gramática establece la estructura de las reglas que se van a aprender a partir de la información proporcionada por el experto:

- Variables de entrada y salida.
- Número de etiquetas lingüísticas de las variables de entrada.
- Estructuras válidas para el consecuente de las reglas.

La gramática está descrita en la Figura A.3. El primer ítem enumera las variables, después los símbolos terminales, en tercer lugar la variable de inicio y finalmente se enumeran las reglas para cada variable. Cuando una variable tiene más de una regla, las reglas se separan mediante el símbolo |. La variable *rule* es la variable de inicio de la gramática y genera dos nuevos nodos en el árbol: *antecedent* y *consequent*. El nodo *antecedent* codifica la parte antecedente de la regla con tres proposiciones (and_j) para cada una de las anteriormente mencionadas variables: largo, ancho y espesor. Cada una de las proposiciones se expresa por medio de una etiqueta lingüística, $A^j_{l_j}$, o por medio de un símbolo λ, que representa que no se ha seleccionado ninguna etiqueta lingüística. Las etiquetas lingüísticas de cada variable de la parte antecedente han sido obtenidas con una partición uniforme del universo de discurso de cada variable de entrada.

La variable *consequent* representa a la parte consecuente de la regla como la suma de las expresiones matemáticas del árbol. Cada variable *expression* representa uno de los elementos del sumatorio de la parte consecuente de la regla definida en la Ecuación A.4: el símbolo terminal α codifica cada uno de los α_i y f_i (Ecuación A.5) y representados por variables *sumExp* y *multExp*, por el símbolo α (la expresión matemática es sólo un coeficiente), o por el símboo λ (esta función no se tiene en cuanta). Finalmente, una expresión matemática puede contener cada una de las variables de entrada (x_j) o saltarse alguna de ellas mediante el símbolo λ ($\delta_{i,\ j} = 0$, ecuaciones A.3, A.5). Esto se representa mediante las reglas de cvar_j.

La Figura A.4 muestra un ejemplo de reglas aprendidas para el algoritmo evolutivo propuesto y esta gramática. La regla tiene tres proposiciones en la parte antecedente (utiliza todas las variables lingüísticas para clasificar las entradas), y dos términos en

- V = { rule, antecedent, ant_{largo}, ant_{ancho}, $\text{ant}_{espesor}$, consequent, expression, sumExp, multExp, cvar_{largo}, cvar_{ancho}, $\text{cvar}_{espesor}$ }
- $\Sigma = \{ \alpha, A_1^{largo}, A_2^{largo}, A_1^{ancho}, A_2^{ancho}, A_1^{espesor}, A_2^{espesor}, x_{largo}, x_{ancho}, x_{espesor}, (,), +, \cdot, \lambda \}$
- S = rule
- Reglas de producción:
  - rule $\longrightarrow$ antecedent consequent
  - antecedent $\longrightarrow$ ant_{largo} ant_{ancho} $\text{ant}_{espesor}$
  - $\text{ant}_{largo} \longrightarrow A_1^{largo} \mid A_2^{largo} \mid \lambda$
  - $\text{ant}_{ancho} \longrightarrow A_1^{ancho} \mid A_2^{ancho} \mid \lambda$
  - $\text{ant}_{espesor} \longrightarrow A_1^{espesor} \mid A_2^{espesor} \mid \lambda$
  - consequent $\longrightarrow$ expression + expression + expression
  - expression $\longrightarrow \alpha \cdot$ (sumExp) $\mid \alpha \cdot$ (multExp) $\mid \alpha \mid \lambda$
  - sumExp $\longrightarrow$ cvar_{largo} + cvar_{ancho} + $\text{cvar}_{espesor}$
  - multExp $\longrightarrow$ $\text{cvar}_{largo} \cdot \text{cvar}_{ancho} \cdot \text{cvar}_{espesor}$
  - $\text{cvar}_{largo} \longrightarrow x_{largo} \mid \lambda$
  - $\text{cvar}_{ancho} \longrightarrow x_{ancho} \mid \lambda$
  - $\text{cvar}_{espesor} \longrightarrow x_{espesor} \mid \lambda$

Figura A.3: Gramática libre de contexto identificar los individuos válidos

el polinomio de la parte consecuente: uno considera la superficie de la pieza y el otro cada una de las dimensiones.

$$\begin{array}{ll} \text{SI} & X_{largo}\ es\ A_1^{largo}\ \text{Y} \\ & X_{ancho}\ es\ A_2^{ancho}\ \text{Y} \\ & X_{espesor}\ es\ A_1^{espesor} \\ \text{ENTONCES} & \text{tiempo es } 120 \cdot \left(x_{largo} \cdot x_{ancho}\right) + \\ & 240 \cdot \left(x_{largo} + x_{ancho} + x_{espesor}\right) \end{array}$$

Figura A.4: Típica regla para la estimación de tiempos de procesado

A.4.2. Algoritmo de programación genética

El algoritmo de programación genética para las estimaciones de tiempos está descrito en la Figurar A.5.

1. Inicializar población
    a) Generar reglas
    b) Evaluar población
    c) Redimensionar población (i)
2. para cada iteración = 1 a *maxIterations*
    a) Cruce y mutación
    b) Evaluar población
    c) Redimensionar población (ii)
3. Seleccionar las reglas para la base de conocimiento final

Figura A.5: Algoritmo evolutivo

A.4.2.1. Generación de la regla inicial

Para cada ejemplo del conjunto de entrenamiento, se genera un individuo (regla) de la siguiente forma: se crea la parte antecedente mediante la selección de las etiquetas que mejor cubren las entradas del ejemplo. Por otro lado, la parte consecuente se obtiene a partir de la variable *consequent* y aplicando aleatoriamente las reglas de producción seleccionadas (reglas de la gramática libre de contexto) recursivamente hasta que todas las hojas del árbol son símbolos terminales.

La Figura A.6 muestra un típico cromosoma correspondiente a la regla representada en la Figura A.4. Los símbolos terminales (hojas del árbol) se representan mediante círculos, y las variables mediante círculos aplanados. A partir del nodo *consequent*, únicamente se puede aplicar una regla. Esta regla genera cinco nodos hijo donde tres de ellos son variables (*expression*). Se pueden seleccionar cuatro reglas de cada uno de ello. Aleatoriamente, se aplicó la regla 2 a la primera variable, la regla 1 a la segunda y la regla 4 a la tercera. El proceso se repitió nuevamente, hasta que las hojas de los árboles son símbolos terminales. El símbolo terminal α puede tomar diferentes valores en cada uno de los nodos donde aparece. Inicialmente, cada uno de los valores asignados a α ha sido obtenido de forma aleatoria. Tras la generación de esta población preliminar, llamada *ejemplos de la población*, los individuos deben de ser evaluados.

A.4.2.2. Evaluación de los individuos

Se repiten los siguientes pasos para cada uno de los individuos y cada uno de los ejemplos del conjunto de entrenamiento:

1. Obtener el grado de cumplimiento de la parte antecedente de la regla (individuo) para el ejemplo.
2. Si la regla cubre el ejemplo:

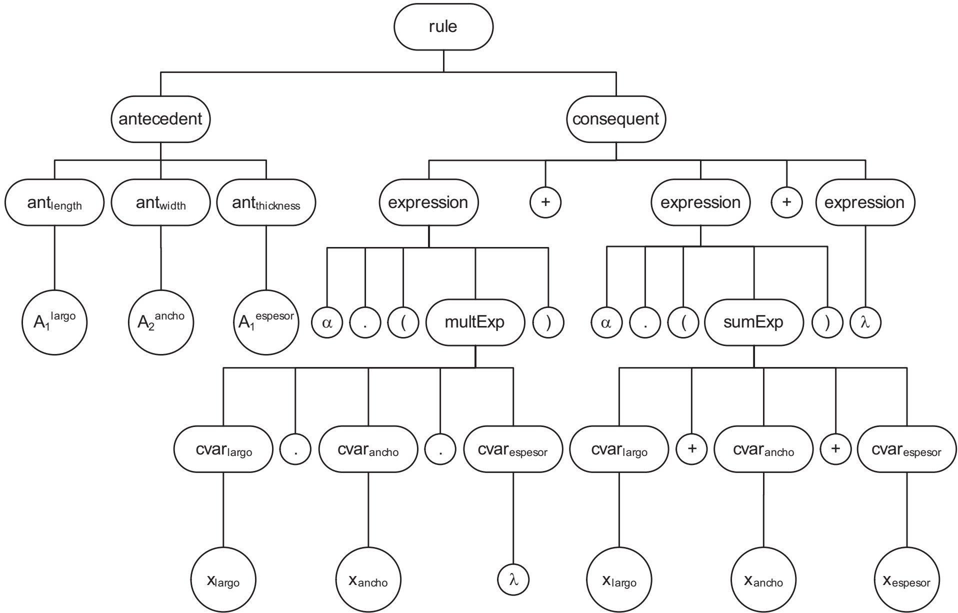


Figura A.6: Ejemplo de cromosoma para el aprendizaje de tiempos de fabricación

a) Analizar sintácticamente la cadena de la parte consecuente de la regla y obtener la estimación de tiempo, $pt_{est}^{c,\ e}$, donde c es el individuo y e el ejemplo.

b) Calcular el error en la estimación de tiempo:

$$error_{c,\ e} = (pt_{est}^{c,\ e} - pt^{e})^2 \tag{A.6}$$

donde pt^e es el tiempo de procesado para el ejemplo e.

Finalmente, es necesario calcular la bondad pura de cada individuo:

$$bondad_{pura}^{c} = \frac{\sum_{e=1}^{ne_c} error_{c,e}}{ne_c} \tag{A.7}$$

donde ne_c es el número de ejemplos cubiertos por el individuo c. La bondad pura mide la fuerza del individuo para los ejemplos que cubre. Para decidir el individuo que aprovecha (del inglés *seize*) cada uno de los ejemplos se utiliza el siguiente algoritmo:

- Para cada ejemplo e, seleccionar el individuo con menor $error_{c,\ e}$
- Si existen varios individuos con igual $error_{c,\ e}$
  - Seleccionar el individuo con mayor $bondad_{pura}^{c}$

- Si existen varios individuos con igual $bondad^c_{pura}$
  - Seleccionar el individuo con mayor ne_c

De esta forma, el individuo que genere un menor error para el ejemplo lo agarrará. Si existen varios individuos con el mismo error, el más fuerte del nicho agarrará el ejemplo. Finalmente, si varios individuos están en esta situación, el que cubra más ejemplos será el seleccionado. La bondad se define de la siguiente manera:

$$bondad^c = bondad^c_{pura} \cdot \frac{aprovechados_c}{cubiertos_c} \tag{A.8}$$

donde $aprovechados_c$ es el número de ejemplos aprovechados por el individuo c y $cubiertos_c$ es el número de ejemplos cubiertos.

A.4.2.3. Ajuste de la población *(i)*

El último paso de la inicialización consiste en eliminar aquellos individuos de la *población de ejemplos* que no cubren ningún ejemplo. Finalmente, pop_{size} individuos son seleccionados de la *población de ejemplos* para construir la población inicial, y dar inicio a la parte iterativa del algoritmo.

La parte iterativa se repite *maxIterations* veces, y se inicial con el cruce y la mutación de los individuos de la población. No existe selección. Se cogen un par de individuos de la población de forma aleatoria (todos los individuos de la población previa deben seleccionarse una vez), estos individuos se cruzan con una probabilidad p_c, luego se mutan con una probabilidad p_m y finalmente se añaden a la nueva población. Al final del proceso la población doblará el tamaño de la población previa, ya que contendrá los individuos originales además de su descendencia (debido al cruce y mutación).

A.4.2.4. Cruce

El cruce de dos individuos se implementa con el operador de cruce en un punto (*point crossover*). El punto de cruce del primer individuo (cp_1) se selecciona aleatoriamente de entre todos los genes del cromosoma (es decir, los nodos del árbol) que son variables de la gramática libre de contexto. Entonces, el algoritmo busca un nodo con la misma variable en el segundo individuo. Si no existe dicho nodo, se selecciona el nodo padre de cp_1 como el nuevo punto de cruce cp_1. Este proceso se repite hasta que exista al menos un nodo en el segundo individuo que sea igual al nodo cp_1. Si existen varios candidatos para seleccionar el cp_2 (punto de cruce del segundo individuo), se selecciona aleatoriamente uno de ellos. Una vez que los puntos cp_1 y cp_2 han sido determinados, el (sub)árbol con el nodo raíz cp_1 se injerta en el nodo cp_2 del segundo cromosoma y viceversa. La Figura A.7 muestra el cruce entre dos individuos. Primero, se selecciona cp_1 en el nodo *sumExp*. Como este nodo no existe en el segundo individuo, el nuevo cp_1 es el nodo padre del nodo con el valor anterior

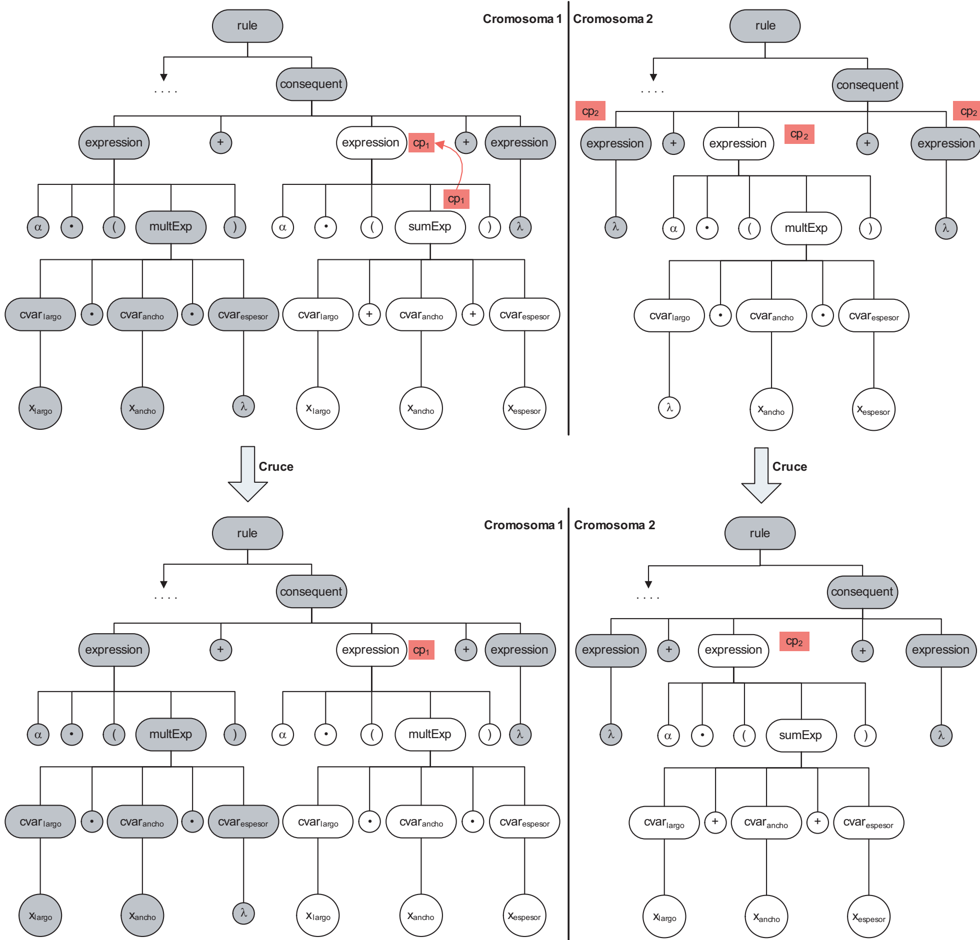


Figura A.7: Ejemplo de cruce de dos cromosomas

de cp_1: el nodo *expression*. Existen tres posibles cp_2, ya que existen tres nodos del mismo tipo que cp_1 en el segundo individuo. Aleatoriamente, el nodo con el fondo en blanco es seleccionado, y los (sub)árboles que tienen como nodo raíz a cp_1 y cp_2 son intercambiados.

A.4.2.5. Mutación

La operación de mutación se inicia seleccionando aleatoriamente un gen. Si el gen es una variable, se aplica aleatoriamente una regla para esta variable y el (sub)árbol resultante reemplaza al original. Si en cambio el gen es un símbolo terminal que puede tomar diferentes valores (α en la gramática), su valor puede mutarse de dos distintas formas: mutación *aleatoria* o mutación *de un paso*. La mutación aleatoria

se selecciona con una probabilidad p_{rm}, y elige un nuevo valor también de forma aleatoria. Por otro lado, la mutación de un paso incrementa o decrementa (con igual probabilidad) el valor del gen en una cantidad llamada $prec_g$ (donde g es el gen), que representa un cambio significativo en el gen.

A.4.2.6. Ajuste de la población *(ii)*

Después del cruce y de la mutación, los individuos se evalúan como en el paso de inicialización. Igualmente, debe ajustarse el tamaño de la población a pop_{size}. Primero, los individuos sin bondad se eliminan. Cuando el tamaño de la población es menor que pop_{size}, se añaden nuevos individuos. Estos individuos se obtienen de la población de ejemplos, seleccionando aquellas reglas que cubren algún ejemplo que no ha sido todavía aprovechado por algún individuo de la población. Si la población sigue estando por debajo de pop_{size} (puede ocurrir en las últimas iteraciones del algoritmo), entonces se insertan copias y copias mutadas de los mejores individuos.

A.4.2.7. Construcción de la base de conocimiento

Una vez que el algoritmo ha finalizado la base de conocimiento estará formada por algunas de las reglas seleccionadas de la población final. Primero, se ordenan de forma decreciente a los individuos en función de su bondad. Partiendo del mejor individuo, se añadirá un individuo si cubre al menos un ejemplo que no está cubierto por reglas ya insertadas en la base de conociminento. Para más información acerca de la solución descrita en este capítulo consultar [189, 188].

A.5. Resultados

La solución propuesta ha sido validada para un conjunto de máquinas típicas del dominio de la fabricación de muebles: cuatro caras (RS-I), lijadora calibradora (ACM), canteadora (VS), escuadradora II (RS-II), y sistema de embalaje de tableros (CSLB). Los ejemplos han sido generados a partir de más de 1.500 piezas de muebles construidas en la fábrica a los largo de varios años. Las dimensiones de cada una de las piezas ha sido obtenida, y para cada una de las máquinas que pueden procesar dicha pieza, se ha medido el tiempo de procesado. Estos tiempos tienen mucho ruido debido a que estas operaciones requieren algún tipo de manipulación por parte de los operadores de la fábrica, y porque el tiempo se mide manualmente.

Cada ejemplo tiene los valores del largo, ancho y espesor de la pieza del mueble, así como el tiempo de procesado de la pieza en la máquina. La Tabla A.3 muestra la media ($\overline{x}$) y la desviación estándar (σ) para cada variable en cada conjunto de datos. El algoritmo debe aprender la base de conocimiento (modelo de regla de la Ecuación A.4) y minimizar el error de la estimación del tiempo de procesado para la máquina, pero manteniendo la estructura de las reglas proporcionadas por el experto y resumidas en la gramática libre de contexto. Los experimentos se han realizado con

Tabla A.3: Características de los conjuntos de datos

Máquina	Largo (m) $\overline{x}$	Largo (m) σ	Ancho (m) $\overline{x}$	Ancho (m) σ	Espesor (m) $\overline{x}$	Espesor (m) σ	Tiempo (s) $\overline{x}$	Tiempo (s) σ
RS-I	1.971	1.132	1.239	0.737	0.251	0.144	1,125	580
ACM	1.971	1.132	1.239	0.737	0.251	0.144	385	265
VS	1.971	1.132	1.239	0.737	0.251	0.144	579	230
RS-II	1.971	1.132	1.239	0.737	0.251	0.144	526	272
CSLB	1.971	1.132	1.239	0.737	0.251	0.144	736	658

una validación cruzada de cinco particiones (*five-fold cross-validation*) para cada uno de los conjuntos de ejemplos. Cada conjunto fue dividido en cinco subconjuntos de igual tamaño, y el proceso de aprendizaje fue ejecutado cinco veces, usando como conjunto de entrenamiento a cuatro de los subconjuntos y como conjunto de pruebas al restante subconjunto. Los conjuntos de prueba son diferentes en cada una de las ejecuciones.

Esta solución ha sido comparada con otras técnicas de regresión. La Tabla A.4 muestra, para cada una de las máquinas (o conjuntos de datos), la media y desviación estándar del número de reglas (#R), el error cuadrático medio del entrenamiento (MSE_{tra}), y el error cuadrático medio de las pruebas (MSE_{tst}) de la validación cruzada de cinco particiones para cada una de las técnicas[1]. En cada tabla, el menor de los valores promedio para #R, MSE_{tra}, y MSE_{tst} están resaltados en negrita. Las metodologías comparadas en las tablas son:

- GP-TSK: es la aproximación descrita en este trabajo. Aprende reglas TSK con una semántica global usando programación genética conjuntamente con una gramática libre de contexto. La diversidad de la población se mantiene mediante *token competition*. El algoritmo ha sido configurado con los siguientes parámetros: $maxIterations = 100$, $pop_{size} = 500$, $p_c = 0{,}8$, $p_m = 0{,}5$ (per chromosome), $p_{rm} = 0{,}25$.
- WM [289]: conocido método ad hoc de Wang and Mendel para la generación de reglas borrosas de tipo Mamdani con una semántica global. Se han usado cinco etiquetas para definir las particiones de cada variable lingüística.
- COR [57, 56]: método ad hoc para el aprendizaje de reglas borrosas de tipo Mamdani con una semántica global. COR tiene dos fases: la construcción del espacio de búsqueda y la selección del conjunto de reglas borrosas más cooperativo. La búsqueda combinatoria para la fase de selección está implementada con un sistema de la mejor-peor hormiga, un algoritmo de optimización de colonias de hormigas. El algoritmo ha sigo ejecutado con los valores estándares y cinco etiquetas para cada variable lingüística.
- COR+TUN [55]: consiste en el previamente descrito algoritmo COR más una fase final de ajuste. Esta fase es un ajunte restringido de los parámetros de la

[1] Los resultados de los métodos WM, MOGUL-TSK, y NN-MPCG se han obtenido usando el software KEEL [7].

función de membresía del algoritmo genético. Se realiza usando intervalos de variación para preservar la semántica de los conjuntos borrosos.

- MOGUL-TSK [6]: algoritmo evolutivo en dos fases basado en MOGUL. La primera fase realiza la identificación local de los prototipos con el fin de obtener un conjunto de reglas TSK inicialmente basadas en una semántica local. Esta fase codifica las reglas mediante la aproximación IRL y se basa en el proceso de generación evolutiva definido en MOGUL. Después, se aplica una fase de post-procesado. Esta fase consiste en un proceso de selección genética de nichos para eliminar reglas redundantes y un proceso genético de ajuste para refinar los parámetros del modelo borroso. El método ha sido ejecutado con los valores estándar y con partición inicial de tres etiquetas para cada variable.

- NN-MPCG [184]: una red neuronal perceptrón multicapa que ajusta sus pesos a través del algoritmo gradiente conjugado. En los experimentos, la red ha sido configurada con una capa oculta y diez neuronas.

A.5.1. Análisis de la precisión

A la vista de estos resultados, el orden de los métodos en función de su precisión es el siguiente: WM, COR, COR+TUN, MOGUL-TSK[2], GP-TSK, y NN-MPCG. Si se comparan los valores medios de MSE_{tst} de la solución propuesta (GP-TSK) con los demás métodos, GP-TSK es el mejor método en dos de las máquinas y el segundo mejor método en las otras tres. Profundizando, las diferencias en cuanto a MSE_{tst} entre GP-TSK y WM van desde el doble para la máquina VS hasta más de siete veces mayor para la máquina ACM. Para COR, MSE_{tst} es más de dos veces mayor en la máquina VS y más de cinco veces mayor para la máquina ACM. De la misma forma, los errores en los test de COR+TUN son un 33 % mayores para la máquina VS y un 153 % mayor que el GP-TSK para la máquina ACM .

La comparación entre MOGUL-TSK y GP-TSK es más informativa, ya que ambos métodos aprenden reglas TSK, aunque con estructuras de consecuente diferentes. GP-TSK obtiene nuevamente mejores valores de MSE_{tst} para todas las máquinas. La diferencias son menores: un 13 % mayores para la máquina ACM mientras que para la máquina VS el error es dos veces mayor.

Finalmente, la aproximación neuronal (NN-MPCG) obtiene los mejores MSE_{tst} en las máquinas RS-I con una mejora del 23 % con respecto a GP-TSK, VS con una mejora del 21 %, y CSLB con una mejora del 38 %. Por el otro lado, en las máquinas ACM y RS-II, la red neuronal tiene errores mayores del 39 % y 4 % respectivamente.

[2] Realmente COR+TUN mejora los valores MSE_{tst} de MOGUL-TSK en tres de los conjuntos de datos de pruebas, pero con un mayor número de reglas. Con un número similar de reglas, MOGUL-TSK es mejor que COR+TUN.

Tabla A.4: Resultados de la validación cruzada de cinco particiones para las distintas máquinas

Máquina	Método	#R		MSE_{tra}		MSE_{tst}	
		$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
RS-I	GP-TSK	**5**	1	5,313	468	5,592	1,113
	WM	125	1	17,827	1,907	19,155	2,360
	COR	96	3	12,547	339	14,113	362
	COR+TUN	96	3	7,029	889	8,277	1,122
	MOGUL-TSK	24	2	5,891	448	6,662	879
	NN-MPCG	—	—	**4,266**	219	**4,297**	387
ACM	GP-TSK	**3**	2	**618**	106	**609**	121
	WM	125	1	4,665	564	4,654	521
	COR	95	3	2,853	112	3,295	221
	COR+TUN	95	3	1,223	88	1,541	297
	MOGUL-TSK	21	3	1,445	418	1,691	761
	NN-MPCG	—	—	831	122	846	169
VS	GP-TSK	**6**	1	1,287	191	1,274	168
	WM	125	1	2,971	88	3,169	401
	COR	102	3	2,172	33	2,479	234
	COR+TUN	102	3	1,376	79	1,691	201
	MOGUL-TSK	23	2	1,303	153	1,440	135
	NN-MPCG	—	—	**990**	35	**1,003**	93
RS-II	GP-TSK	**5**	1	**1,088**	113	**1,096**	189
	WM	125	1	4,811	398	4,980	718
	COR	94	4	3,066	77	3,601	374
	COR+TUN	94	5	1,511	97	1,953	264
	MOGUL-TSK	20	3	2,363	861	2,729	1,001
	NN-MPCG	—	—	1,096	81	1,139	153
CSLB	GP-TSK	**3**	1	4,192	363	4,545	1,301
	WM	125	1	18,860	2,006	19,731	2,217
	COR	105	3	11,755	360	14,852	1,276
	COR+TUN	105	3	5,167	761	7,112	980
	MOGUL-TSK	21	2	8,407	2,189	9,612	2,034
	NN-MPCG	—	—	**2,723**	135	**2,816**	624

A.5.2. Discusión acerca de la estructura del conocimiento

Como ya hemos mencionado, mantener la estructura del conocimiento proporcionado por el experto es de gran importancia en este sistema. Esto se debe a que los expertos necesitan analizar las funciones de regresión que son generadas por las estimaciones del tiempo de procesado. Con esta información, el experto debería ser capaz de modificar un plan de producción o simplemente rechazar alguna parte del plan. Se pueden distinguir cuatro distintos niveles de similaridad entre la información que puede ser extraída por el experto. El nivel más bajo se corresponde con la aproximación neuronal NN-MPCG: el experto no tiene información acerca de cómo un tiempo ha sido estimado. Los métodos WM, COR y COR+TUN proporcionan un poco más de información: "si la longitud es pequeña y el ancho es pequeño entonces el tiempo es pequeño ". Con esta regla el experto puede extraer que la variable *espesor* no afecta la estimación en esa máquina y también que las piezas con un largo y ancho pequeños generarán tiempos de procesado pequeños. Sin embargo, el experto no puede deducir

1. SI longitud es grande Y ancho es pequeño Y espesor es grande ENTONCES tiempo = 120 · longitud + 60 · ancho + 180 · (longitud · ancho · espesor)
2. SI longitud es pequeño ENTONCES tiempo = 120 · longitud + 240 · (longitud · ancho · espesor)
3. SI ancho es pequeño Y espesor es pequeño ENTONCES tiempo = 120 · longitud + 60 · espesor + 180 · (longitud · ancho · espesor)
4. SI ancho es pequeño ENTONCES tiempo = 120 · longitud + 240 · (longitud · ancho · espesor)

Figura A.8: Típica base de reglas para la máquina ACM

la contribución de cada una de las variables a la estimación de tiempos.

Este problema está parcialmente resuelto en MOGUL-TSK, ya que el experto conoce la contribución de cada una de las variables de entrada en la estimación de tiempo: "si longitud es pequeña entonces 50 + 100 · largo + 300 · ancho ". Los consecuentes de MOGUL-TSK son polinomios de primer orden, con lo cual no es posible extraer a través de las reglas las relaciones entre las distintas variables (por ejemplo un polinomio que depende de la superficie o del volumen). Además, MOGUL-TSK genera bases de conocimiento con una semántica local, mientras que GP-TSK tiene una semántica global. Es más, las reglas de GP-TSK también aporta información acerca de las relaciones y variables, ya que pueden generarse los polinomios de orden superior a través de la gramática: "si la longitud es pequeña y el ancho es pequeño entonces el tiempo es 60 · (largo · ancho) ". Tanto esta regla como la generada por MOGUL-TSK estiman el tiempo de procesado mediante las variables *largo* y *ancho*. Sin embargo, una regla GP-TSK aporta información acerca de la relación entre ambas variables, generando una nueva variable para el experto: la superficie de la pieza (*largo* × *ancho*). Además, la aproximación GP-TSK elimina la existencia de reglas difíciles de interpretar por parte del experto. Por ejemplo, MOGUL-TSK podría generar una regla con el cuadrado del largo en la parte consecuente (si se utiliza una aproximación de orden superior) pero en cambio GP-TSK restringe la composición de las variables a la gramática libre de contexto. Finalmente, GP-TSK obtiene un número pequeño de reglas (entre 3 y 6), mientras que MOGUL-TSK obtiene entre 20 y 24 reglas en las distintas máquinas. Un número reducido de reglas también ayuda al experto a extraer la información de la función de regresión, ya que implica la ejecución de un número menor de reglas.

La Figura A.8 muestra a título de ejemplo una base de reglas típica generada por GP-TSK. Claramente esta base de reglas satisface las dos premisas de este desarrollo:

- Simplicidad para la extracción de información: la estructura de reglas cumple los requerimientos del experto. Las características que facilitan la extracción de información de la base de conocimiento son:
  - La contribución de cada variable de entrada es explícita.
  - Las reglas representan la asociación de las variables (como la superficie o el volumen), facilitan la comprensión de las estimaciones de tiempo además

de evitar la generación de nuevas variables sin un significado real para el experto.

- El número de reglas es reducido.

- La precisión de las estimaciones de tiempo obtenidas por GP-TSK ha demostrado ser muy elevada, y es únicamente superada por la aproximación neuronal en tres de los cinco conjuntos de datos. En cualquier caso, desde la perspectiva del experto, la aproximación neuronal no es aceptable, debido a la imposibilidad de extraer información de sus estimaciones.

B

Planificación de la producción

La planificación de la producción es un problema de gran complejidad en entornos industriales como la fabricación de muebles a medida. Este problema se define como la búsqueda de la secuencia óptima de operaciones a realizar a partir de un conjunto recursos y restricciones. Este tipo de planificación está clasificado como un problema NP-duro [111] debido a que genera un espacio de búsqueda combinatorio: tiene que combinar la asignación de recursos para maximizar la ocupación de las máquinas y para minimizar el tiempo requerido para procesar el plan de trabajo. Debido a estas características, no es aconsejable resolver este problema a través de métodos exactos como son *branch and bound*, programación dinámica o programación con restricciones. En estas situaciones, las soluciones próximas al óptimo se consideran buenas soluciones, y por ello, métodos como los algoritmos evolutivos, simulated annealing, tabu search o de investigación operativa (como el simplex) están mejor adaptados este a esta problemática [13].

El problema de la planificación de la producción se caracteriza por la presencia de muchos objetivos contradictorios. Por ello, es natural mirar a este tipo de planificación como a un problema de optimización multi-objetivo que trata de combinar diferentes objetivos con el fin de obtener la mejor solución posible. La selección de un método de optimización debe fundamentarse en las características del problema a resolver, en nuestro caso debía de permitir resolver un problema real de planificación encuadrado dentro de la familia de problemas denominados *job-shop scheduling problems*. Este tipo de problema se caracteriza por un espacio de búsqueda enorme que inclusive podría ser mayor si hubiésemos tenido en cuanta eventos como la rotura de máquinas, las bajas o la aparición de nuevos pedidos. La planificación que se presenta en este apéndice tratará estos nuevos eventos como una nueva planificación y no como un ajuste de la planificación actual, es decir, sólo afectarán las condiciones de la nueva planificación a realizar. Por esta razón, además de la fiabilidad propia de cualquier planificación, nuestra aproximación requiere eficiencia, ya que el tiempo es un factor crucial para dar una respuesta ágil a los nuevos pedidos. En este contexto, el uso de métodos basados en algoritmos evolutivos ha demostrado ser una solución rápida y robusta en problemas de optimización como es la planificación de la producción. Estos métodos facilitan encontrar óptimos globales y no quedarse atrapado en óptimos locales, aspecto que no aseguran los métodos de gradiente [74, 44].

En este apartado describiremos dos métodos primitivos que dan soporte a la tarea de planificación de la carga de producción que forma parte del WF que hemos descrito en el Capítulo 7. El primero de ellos[1] [284, 283] se aplicará a planificaciones que parten de un diseño conceptual poco detallado donde la incertidumbre respecto al mueble a fabricar es importante. El segundo método[2] [281, 282] permite obtener una planificación más completa y precisa. Este método, además de aplicarse en la elaboración de presupuestos, también se utiliza para obtener diariamente el plan de trabajo a realizar en una ventana temporal de dos semanas.

B.1. Características del problema de planificación

El problema de planificación consiste en encontrar un plan de trabajo que satisfaga un conjunto de restricciones. En este contexto, un plan de trabajo es la asignación de intervalos de tiempo a las m máquinas $M = \{M_1, M_2, \ldots, M_m\}$ que realizan los n trabajos $J = \{J_1, J_2, \ldots, J_n\}$ del problema. Cada trabajo $J_i \in J$ consiste en un conjunto de n_i operaciones $O_i = \{O_{i,1}, O_{i,2}, \ldots, O_{i,n_i}\}$ con un tiempo de procesado $p_{k,i,j}$ en una determinada máquina M_k. Por ejemplo, el plan de trabajo puede representarse a través de un diagrama de Gantt como puede verse en la Figura B.1. Cada una de las filas de la figura representa una máquina y cada una de las cajas del diagrama representa una operación con un determinado intervalo de tiempo.

Un plan de trabajo representa la asignación en el tiempo de de los recursos a los trabajos. Por ello, los planes de trabajo que satisfacen todas las restricciones se denominan planes de trabajo *viables*. Este tipo de planes de trabajo se puede clasificar como:

- *Planes de trabajo semi-activos*: cuando ninguna operación puede iniciarse antes sin cambiar el orden de procesamiento o violar alguna restricción.
- *Planes de trabajo activos*: cuando ninguna operación puede iniciarse antes sin retrasar al menos otra operación o violar alguna restricción.
- *Planes de trabajo sin-retrasos*: cuando ninguna máquina está inactiva si una operación está lista para ser procesada.

La relación entre estas clases de planes de trabajo y los planes de trabajo óptimos está representada en la Figura B.2. Habitualmente, los problemas de planificación únicamente buscan entre el conjunto de planes de trabajo activos, ya que permiten reducir considerablemente el espacio de búsqueda y siguen garantizando poder encontrar una posible solución óptima.

[1] Este trabajo recibió el segundo premio al *best industry-orientated paper* en el 2006 International Symposium on Evolving Fuzzy Systems (EFS'06), celebrado en Lake District, Reino Unido.

[2] Este trabajo recibió el premio al *industry best paper application award* en el 8th International Conference on Hybrid Intelligent Systems (HIS'08), celebrado en Barcelona, España.

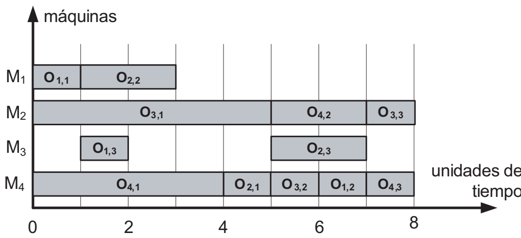


Figura B.1: Diagrama Gantt de un plan de trabajo. El símbolo $O_{i,j}$ representa la operación j del trabajo J_i.

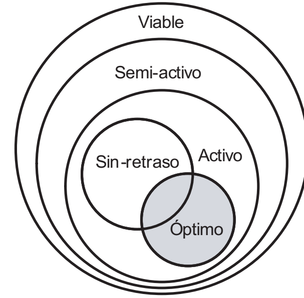


Figura B.2: Diagram de Venn para los diferentes tipos de planes de trabajo

La planificación de la producción es dependiente de las características del producto a fabricar, de las máquinas encargadas de la fabricación y de los criterios de optimalidad [253]. En cuanto a las características del trabajo, éste puede tener:

- *Derecho a interrumpirse*. Un trabajo puede ser interrumpido y resumido posteriormente, incluso en una máquina diferente.
- *Relaciones de precedencia*. La precedencia entre trabajos puede representarse mediante grafos acíclicos o árboles.
- *Fecha de salida*. Cada máquina M_k tiene un tiempo $s_{k,i,j}$ antes del cual ningún procesado puede realizarse en dicha máquina para la operación $O_{i,j}$.
- *Restricciones acerca del tiempo de procesamiento*. Por ejemplo, se puede establecer un *tiempo unidad* $p_{i,j} = 1$ para el procesamiento de todas las operaciones de un determinado trabajo $O_{i,j}$.
- *Fechas de entrega*. Un trabajo J_i debe finalizarse antes de una determinada fecha d_i.
- *Operaciones agrupadas*. Un conjunto de operaciones pueden agruparse de forma que puedan lanzarse conjuntamente en una determinada máquina.

La configuración de las máquinas define el tipo de problema de planificación. Los primeros cuatro problemas de la siguiente lista se refieren a modelos de una operación (trabajos compuestos por una única operación) mientras que los cinco últimos se refieren a modelos multi-operación:

- *Máquinas dedicadas*. Cada trabajo debe procesarse en una máquina dedicada.
- *Máquinas paralelas e idénticas*. Todas las máquinas que procesan un trabajo son idénticas. Entonces, tienen el mismo tiempo de procesado, $p_{k,i,1} = p_i$.

- *Máquinas paralelas y uniformes.* El tiempo de procesado de la máquina M_k es $p_{k,i,1} = p_i/s_k$ donde s_k es la velocidad de la máquina.
- *Máquinas paralelas no relacionadas.* El tiempo de procesado de una máquina M_k depende del trabajo a realizar, $p_{k,i,1} = p_i/s_{i,k}$.
- *General-shop.* Las máquinas están dedicadas y existen relaciones de precedencia entre las operaciones.
- *Open-shop.* Equivalente a la planificación *general-shop* exceptuando que no existen relaciones de precedencia. Las operaciones de un trabajo no tienen porque procesarse en un determinado orden.
- *Job-shop.* Caso especial de la planificación *general-shop* donde la relación de precedencia indica que la operación $O_{i,j}$ es la operación j-ésima del trabajo i y no puede comenzar hasta que la operación $O_{i,j-1}$ haya finalizado, $1 < j < n$.
- *Flow-shop.* Caso especial del problema *job-shop* donde el orden de procesamiento de las operaciones O_i es el mismo para todos los trabajos. Ello no implica que los trabajos sean iguales, ya que su tiempo de procesado puede variar.
- *Mixed-shop.* Una combinación de planificación *job-shop* y *open-shop*.

También existen múltiples criterios de optimalidad para los problemas de planificación. Las medidas de rendimiento más usadas son:

- *Makespan.* El tiempo máximo de compleción (longitud del plan de trabajo).
- *Total flow-time.* El tiempo total de procesamiento de todos los trabajos.
- *Total lateness.* La suma del tiempo con respecto a la fecha de entrega de cada trabajo. Se tienen en cuenta tanto entregas anticipadas como retrasadas.
- *Total tardiness.* La suma del tiempo con respecto a la fecha de entrega de cada trabajo. Sólo se tienen en cuenta las entregas retrasadas.
- *Total earliness.* La suma del tiempo con respecto a la fecha de entrega de cada trabajo. Sólo se tienen en cuenta las entregas anticipadas.
- *Maximum lateness.* El mayor tiempo de retraso.
- *Maximum tardiness.* El mayor tiempo de retraso sin tener en cuenta entregas anticipadas.

La planificación resuelta en este trabajo se clasifica dentro de los problemas de la clase job-shop scheduling (JSSP). De entre los distintos tipos de problemas de planificación, el JSSP es uno de los más desafiantes: problemas de un tamaño $n \times m$ y $m \geq 2$ son NP-difícil (NP-hard) y está considerado como el peor tipo de problema combinatorio al que enfrentarse [111]. El espacio de búsqueda para el JSSP tradicional es $(n!)^m$.

B.2. Aproximaciones tradicionales

Se han propuesto muchas aproximaciones para resolver el JSSP. Algunas de estas aproximaciones han optado por métodos exactos. Por ejemplo en [46, 51], se utilizan versiones perfeccionadas del método *Branch and Bound* (BB) para minimizar el makespan en el problema clásico. Aunque estos métodos han demostrado ser muy útiles en problemas de tamaño pequeño y medio, su excesivo tiempo de computación hace que sean poco viables en problemas de gran tamaño [148]. Otras aproximaciones exactas del campo de la investigación operativa, como la programación lineal [22] o la programación dinámica [21] también se han aplicado al JSSP. Sin embargo, el número de restricciones y/o variables para dar soporte a este tipo de problemas es muy grande incluso en problemas de tamaño pequeño, y por ello estas técnicas no son efectivas para problemas de gran tamaño.

La mayoría de las aproximaciones han optado por métodos heurísticos en búsqueda de soluciones que incluso puedan no ser óptimas pero que aseguren un tiempo de computación razonable. Sistemas basados en conocimiento [315], en reglas de producción [242] o en redes neuronales también se han utilizado para resolver los JSSPs. De entre este conjunto de técnicas, los algoritmos meta-heurísticos han demostrado un mayor rendimiento. Por ejemplo, el *Tabu Search* (TS) ha demostrado ser muy efectivo para resolver el JSSP [291, 194]. Sin embargo, cuando se aplica a problemas de optimización complejos, como lo es un problema de planificación del mundo real, el rendimiento del algoritmo depende de la solución inicial empleada para la optimización. Otra técnica muy empleada en los JSSPs se denomina *Simulated Annealing* (SA) [265, 2]. SA permite evitar máximos y mínimos locales, pero no consigue obtener buenas soluciones rápidamente. Para mitigar este problema, el SA se suele combinar con el TS. Otro ejemplo es la heurística *Shifting Bottleneck* (SB). Esta heurística parte el JSSP en un conjunto de planificaciones a una máquina. Esta heurística está considerada como el primer método que aproximó eficientemente la resolución del JSSP [3, 21]. Para conseguir este rendimiento, el SB requiere de un importante esfuerzo computacional debido a las re-optimizaciones a aplicar para obtener un buen resultado.

Los Algoritmos Evolutivos (EA) son en la actualidad una de las mejores apuestas para resolver este tipo de problema. Es más, han demostrado tener un mayor rendimiento que los métodos tradicionales y heurísticos cuando se aplican al JSSP [128]. Muchos de estos EAs se han aplicado a la resolución del problema clásico o con pequeñas variaciones. Difieren entre ellos principalmente en el esquema de representación del cromosoma, en los operadores, en los niveles de hibridación con otras heurísticas y en las medidas de rendimiento aplicadas.

Existen muchos esquemas de representación del cromosoma recogidos en la bibliografía de los JSSP [204]. Los cromosomas pueden utilizar una representación directa o indirecta. En la representación directa, el plan de trabajo se codifica en el propio cromosoma. Por ejemplo, los cromosomas basados en la operación codifican el plan de trabajo como secuencias de operaciones y donde cada gen representa una operación. Aunque la representación de estos cromosomas es relativamente sencilla, se

requieren operadores de cruce y mutación complejos. En cambio, la representación indirecta facilita la definición de operadores simples pero el cromosoma deja de ser tan sencillo. En esta representación el cromosoma representa directamente un plan de trabajo viable. Por ejemplo, cada gen del cromosoma puede representar una regla de asignación.

Una de las ventajas de los EA es que han incorporado criterios multi-objetivo a sus modelos [251, 125]. La mayoría de las aproximaciones se fundamentan en la combinación de múltiples objetivos en un único objetivo escalar combinado a través de coeficientes [42, 166]. Sin embargo, los EA multi-objetivos (MOEAs) actuales utilizan métodos de selección y sustitución basados en criterios de dominación multi-objetivo (dominación Pareto). Ejemplos de esta aproximación son los MOGA [108], NPGA [136], PESA [78], SPEA [312] y NSGA-II [85]. Por ejemplo, en [61] se aplica un MOEA a la planificación de operaciones de perforación en circuitos integrados de cara a minimizar el makespan y el total flow time. En [245] se aplica al problema JSSP clásico con las medidas de rendimiento makespan y total tardiness. En [237] a la fabricación de móviles con tres objetivos: *makespan*, *total flow time* y *machine idleness*. En [205] se aplica un MOEA para derivar la secuencia óptima de reglas de asignación en el algoritmo de *Giffer and Thompson* y en [304] al JSSP flexible.

Finalmente mencionar otras aproximaciones meta-heurísticas que se han aplicado al JSSP: *Greedy Randomized Adaptive Search Procedure* (GRASP) [32], colonias de hormigas y colonias de abejas [172, 65], *Jumping Genes Genetic Algorithm* [231], etc. Sin embargo, existen pocos resultados disponibles de la aplicación de estas técnicas y todavía no está comprobado que obtengan mejores resultados que los algoritmos actuales [128].

B.3. Adaptación del JSSP a la fabricación de muebles a medida

El tiempo de fabricación se define como el intervalo que transcurre mientras la planta de fabricación realiza todas las operaciones necesarias para completar una orden de trabajo. Una estimación precisa de este tiempo es muy complicada en industrias donde dominan los productos hechos a medida, como es el caso de la industria del mueble. La falta de experiencias de fabricación previas y la variedad en la producción hacen que la obtención de estas estimaciones sea compleja.

La reducción del tiempo de fabricación conlleva muchos beneficios: reducción de costes de almacenaje, incremento de la calidad del producto (los problemas en el proceso son analizados previamente), respuesta más eficaz a las órdenes de pedido de los clientes, e incremento en la flexibilidad. Cualquier empresa dedica una parte importante de su esfuerzo en la reducción del tiempo de fabricación a través de la mejora y organización de sus procesos de producción, de sus sistemas de control o mediante el desarrollo de procedimientos más sofisticados de planificación [131, 161]. En este último caso, el objetivo es la definición de planes de trabajo que permitan minimizar la cola de trabajo de los recursos y maximizar su capacidad, teniendo

en cuenta la disponibilidad del material y los requisitos del producto. Para ello, se analizan la viabilidad, el tiempo y el coste de los planes más prometedores [123, 182]. Cabe mencionar que el tiempo de fabricación tiene muchas componentes [8, 63, 230] (movimiento, cola, configuración y tiempos de procesado) y en este trabajo se hace uso de todos ellos para mejorar la planificación y producir planes de trabajo viables.

La planificación en la industria del mueble es similar a la realizada en el JSSP clásico:

- Cada trabajo J_i define unas relaciones de precedencia para sus operaciones $O_{i,j}$.
- Cada trabajo J_i debe finalizarse antes de una fecha límite d_i.
- No existen restricciones de procesado en las operaciones.
- Aunque algunas operaciones de esta industria podrían considerarse como agrupaciones, en esta planificación se tratan como operaciones simples.
- Cada máquina tiene un tiempo de inicio $s_{k,i,j}$ antes del cual la máquina no realizará procesado para la operación $O_{i,j}$.
- Cada máquina puede procesar únicamente una operación a la vez.
- Ninguna máquina puede liberar una operación hasta que ésta haya finalizado.
- El número de máquinas de cada tipo está predefinido antes de la planificación y es mayor que uno.
- Ningún trabajo puede iniciarse antes de estar disponibles las piezas que se van a procesar.

Sin embargo, algunas restricciones típicas de esta industria modifican las características típicas de un JSSP clásico y lo acercan a un JSSP flexible:

- Dos operaciones del mismo trabajo pueden ser procesadas simultáneamente. En esta planificación las relaciones de precedencia se representan a través de grafos acíclicos mientras que en el JSSP clásico se representan mediante una secuencia de operaciones. La industria del mueble, al igual que muchas otras, fabrica productos hechos de muchas clases de materiales. Por ejemplo, la mesa de la Figura B.3 está hecha de tablero rechapado con madera, acero y cristal. Por ello, el procesado de los distintos materiales, hasta su ensamblado, puede realizarse simultáneamente. Las relaciones de precedencia se fundamentan en el conocimiento propio de de la fabricación de muebles. Estas relaciones consideran tanto aspectos constructivos, como son las uniones que ensamblarán las distintas piezas del mueble, como aspectos relacionados con el acabado o los estándares de calidad.
- Cada trabajo puede planificarse en la misma máquina más de una vez. En el JSSP clásico se requiere que la secuencia de operaciones $O_{i,j}$ del trabajo J_i

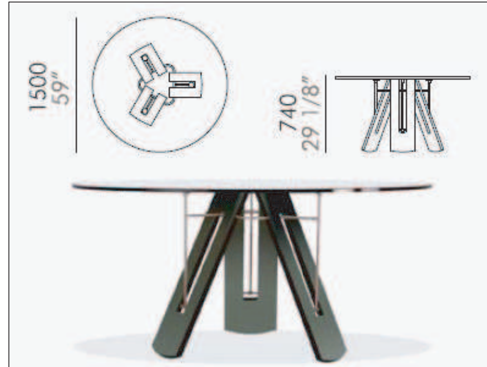


Figura B.3: Diseño de una mesa de oficina. Las piezas, los procesos de fabricación y los acabados se pueden extraer a partir de este diseño.

pase por cada una de las máquinas exactamente una vez. Sin embargo, cada máquina puede realizar procesados distintos en el entorno de fabricación típico de una fábrica de muebles. Por ejemplo, la máquina canteadora puede usarse para cantear, recortar o lijar. Por ello, la canteadora podría procesar distintas operaciones de un mismo trabajo.

- Los trabajos no tienen porqué visitar cada una de las máquinas en M. Aunque en el JSSP clásico se requiere la visita de cada una de las máquinas, el tipo de fabricación llevado a cabo en una fábrica de muebles no permite cumplir este requerimiento. Por ejemplo, una mesa hecha de madera maciza nunca tendrá la misma secuencia de operaciones que la mesa compuesta de metal y cristal representada en la Figura B.3.

B.4. Método primitivo de planificación JSSP con máquinas dedicadas

En ciertas ocasiones no es posible diseñar con precisión los detalles de un mueble en fase de presupuestado. Esta situación suele producirse cuando se quiere obtener una idea orientativa del mueble a fabricar o bien cuando no existe tiempo material para crear los planes CAD. En cualquiera de los casos, no es recomendable aplicar una planificación detallada, ya que ésta fundamenta sus cálculos en información que no estará presente en el plano del mueble.

El método que describimos en este apartado trata el problema de planificación para situaciones con un mayor grado de incertidumbre. Además, este tipo de planificación suele estimar los tiempos de procesado a través de fórmulas orientadas a la obtención de un tiempo para el presupuestado más que a la obtención de un tiempo real de producción. El algoritmo que realiza la búsqueda de las mejores asignaciones

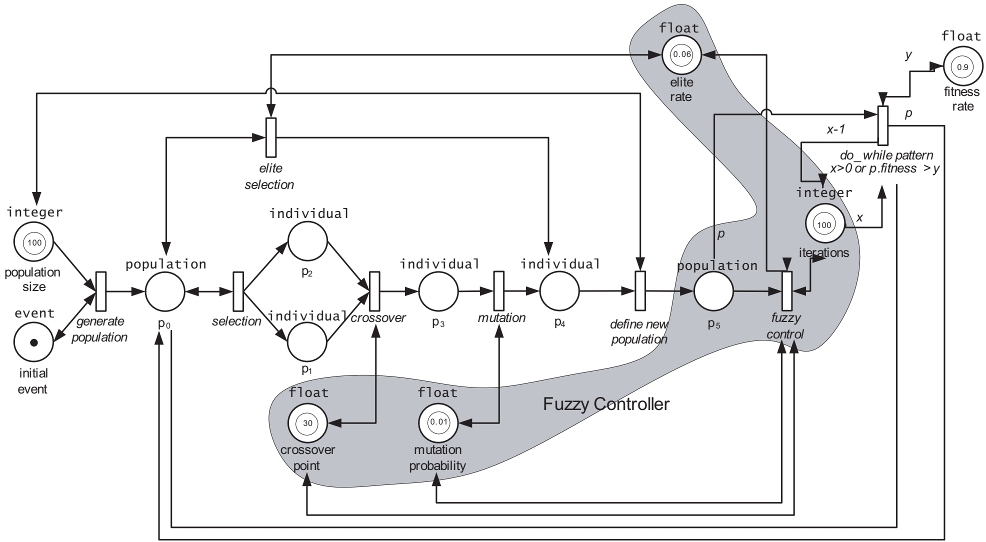


Figura B.4: Representación en forma de red de Petri del método primitivo que resuelve el problema de planificación JSSP con máquinas dedicadas

de recursos está representado en la Figura B.4. En este caso, se ha enriquecido un algoritmo genético clásico con un controlador borroso. En la industria del mueble a medida, la estimación de tiempos y la carga de fabricación prevista suelen basarse en la experiencia, en datos empíricos y en proyecciones y, por lo tanto, sus resultados tienen cierto grado de incertidumbre. Por ello, es necesario controlar el grado de confianza de las soluciones obtenidas por el algoritmo evolutivo. En esta aplicación, un controlador borroso se encarga de supervisar las nuevas poblaciones (soluciones de la búsqueda) modificando los ratios de elitismo, cruce y mutación de cara a incrementar la calidad de las soluciones y evitar convergencias prematuras. La evaluación borrosa de la población y el control del algoritmo evolutivo se realiza en varios pasos:

- En una fase previa, los expertos han definido las etiquetas lingüísticas que establecen la calidad de las estimaciones de tiempo realizadas por las funciones de regresión. Esta fase consiste en asignar una etiqueta (ALTO, MEDIO, BAJO) a la estimación temporal y de esa forma definir su incertidumbre.
- Se colecta la información histórica necesaria para comparar las estimaciones de tiempo con los datos reales de la producción. Esto permite al sistema calcular la media de error $\Delta p_{k,i,j}$ para cada una de las estimaciones de tiempo $p_{k,i,j}$. Este error es la clave para calcular la fiabilidad de las estimaciones futuras. Si no existe un histórico de tiempos, entonces la fiabilidad se interpola a partir del propio $p_{k,i,j}$.
- La primera etapa evalúa la calidad de las estimaciones de tiempo de cada una de las operaciones $O_{i,j}$ para una máquina M_k usando la información histórica

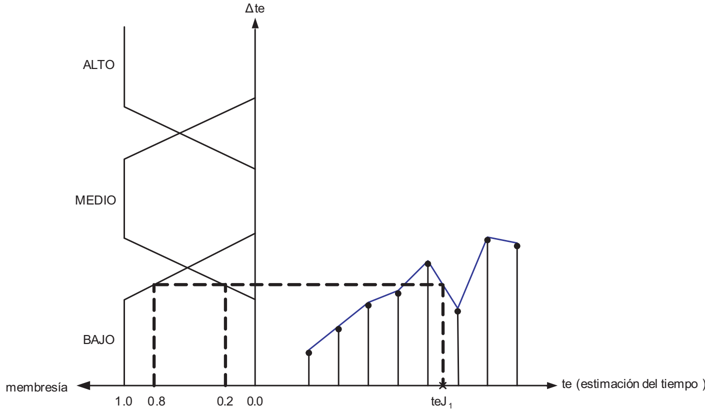


Figura B.5: Cálculo de la incertidumbre de una operación

ya obtenida. El porcentaje de error $\Delta p_{k,i,j}$ para cada operación se obtiene a partir de su grado de pertenencia a cada una de las etiquetas lingüísticas que indican su calidad.

- El segundo paso calcula la calidad de todo el plan: los grados de incertidumbre/pertenencia calculados para cada una de las operaciones se agregan por medio de una t-conorma.

- Finalmente, tanto las estimaciones de tiempos del plan de trabajo como las medidas de incertidumbre:

  1. Se utilizan como entrada del controlador borroso para mejorar la planificación del algoritmo evolutivo.
  2. Se realimentan al experto como indicación de la calidad del plan. Para facilitar su comprensión, se aproxima dicha valoración a través de una etiqueta lingüística.

La Figura B.5 describe gráficamente el proceso. El histórico porcentual de desviaciones de las estimaciones de tiempo de una operación se representan en el eje $p - \Delta p$, mientras que las variables lingüísticas definidas por el experto se representan en el eje $membership - \Delta p$. Para una nueva operación $O_{i,j}$ y máquina M_k, se calcula su estimación de tiempo $p_{k,i,j}$ y su correspondiente porcentaje medio de error $\Delta p_{k,i,j}$ e incertidumbre. Para el ejemplo representado en la Figura B.5, la incertidumbre de la operación $O_{i,j}$ y máquina M_k es $(BAJO_{0,8},\ MEDIO_{0,2},\ ALTO_{0,0})$. Para un plan que incluya una misma operación $O_{i,j}$ para las máquinas M_k y M_{k+1} con $(BAJO_{0,0},\ MEDIO_{0,4},\ ALTO_{0,6})$, entonces la incertidumbre total asociada al plan vendría dada por $(BAJO_{s(0,8,0,0)},\ MEDIO_{s(0,2,0,4)},\ ALTO_{s(0,0,0,6)})$, donde s es el operador t-conorma usado para la agregación. Para más información acerca del algoritmo evolutivo y del controlador borroso consultar [284, 283].

B.5. Método primitivo de planificación JSSP

Un problema multi-objetivo de planificación se puede describir como un problema multi-objetivo de optimización:

$$min\ F(x) = \{f_1(x), f_2(x), ..., f_k(x)\}\ \ s.\ t.\ x \in S$$

donde x es la solución, S es el conjunto de soluciones viables, k es el número de objetivos del problema, $F(x)$ es la imagen de x en el espacio de k-objetivos y cada $f_i(x)$ para $i = 1, \ldots, k$ representa un objetivo. En la fabricación de muebles a medida, al igual que en muchas otras fabricaciones del mundo real, existen $f_i(x)$ objetivos en conflicto. Por ejemplo, objetivos como *minimizar el coste de fabricación* y *minimizar el tiempo de fabricación* pueden entrar en conflicto, ya que normalmente las máquinas más rápidas suelen tener un mayor coste de amortización. Por lo tanto y en contraste con los problemas con un único objetivo, no existe una mejor solución sino un conjunto de soluciones que seleccionar (soluciones no dominadas).

La aproximación tomada en este trabajo realiza la optimización multi-objetivo a través del algoritmo NSGA-II [85]. El NSGA-II es uno de los MOEAs más eficientes: su complejidad computacional es $O\left(MN^2\right)$, donde M es el número de objetivos y N es el tamaño de la población. La estructura del algoritmo se describe en la Figura B.6. NSGA-II tiene un esquema de asignación de bondad que consiste en ordenar la población en diferentes frentes utilizando la relación de no dominación. Por lo tanto, tiene dos objetivos: *(i)* encontrar un conjunto de soluciones no dominadas los más cerca posible del frente de Pareto óptimo en cada iteración y *(ii)* mantener el conjunto de soluciones lo más diversificado posible de forma que cubra o se acerque al frente de Pareto óptimo.

El bucle principal del algoritmo comienza con la combinación de la población actual y previa, y el cálculo de los frentes no dominados de R_t. La función *fast-non-dominated-sort* se encarga de este paso. En un primer paso, para cada individuo p de la población P, se realizan dos cálculos: el conjunto de individuos dominados por p (S_p), y el número de individuos que dominan a p (n_p). Todos estos individuos (p) no dominados tendrán un rango $p_{rank} = 1$ y pertenecerán al primer frente del Pareto (frente no dominado) (V_1). Para calcular los demás frentes del Pareto (paso 3), para cada uno de los individuos (p) del frente anterior, cada elemento q del conjunto S_p reduce su contador de dominación en 1 (n_q). De esta forma, todos estos individuos q con $n_q = 0$ pertenecerán al frente Pareto i-ésimo.

Una vez que todos los frentes de Pareto han sido determinados, el bucle principal del algoritmo añade (paso 4) a la nueva población (P_{t+1}) todos los individuos en el frente i-ésimo, empezando por $i = 1$ e incrementando i mientras el tamaño de la población sea menor que N. Además, también se calcula una medida de la distancia de multitud (función *crowding-distance-assignment*) a cada uno de los individuos pertenecientes a V_i. Esta distancia es la suma de las distancias de multitud de cada uno de los objetivos y aporta una estimación de la densidad y soluciones del problema.

Bucle principal

1. $R_t = P_t \bigcup Q_t$
2. $V = \textit{fast-non-dominated-sort}\,(R_t)$
3. $P_{t+1} = \emptyset$ and $i = 1$
4. *repetir hasta* $|P_{t+1}| + |V_i| \leq N$
   $\textit{crowding-distance-assignment}\,(V_i)$
   $P_{t+1} = P_{t+1} \bigcup V_i$
   $i = i + 1$
5. $\textit{ordenar}\,(V_i,\ \prec_n)$
6. $P_{t+1} = P_{t+1} \bigcup V_i\,[1 : (N - |P_{t+1}|)]$
7. $Q_{t+1} = \textit{crear-nueva-población}\,(P_{t+1})$
8. $t = t + 1$

$\textit{crowding-distance-assignment}\,(I)$

1. $l = |I|$
2. *para cada* i
   $I\,[i]_{dist} = 0$
3. *para cada objetivo* m
   $I = \textit{ordenar}\,(I,\ m)$
   $I\,[1]_{dist} = I\,[l]_{dist} = \infty$
   for $i = 2$ *to* $l - 1$
   $I\,[i]_{dist} = I\,[i]_{dist} +$
   $(I\,[i+1]\,.\,m - I\,[i-1]\,.\,m)\,/$
   $\left(f_m^{max}\,(x) - f_m^{min}\,(x)\right)$

$\textit{fast-non-dominated-sort}\,(P)$

1. *para cada* $p \in P$
   $S_p = \emptyset,\ n_p = 0$
   para cada $q \in P$
   si $(p \prec q)$ *entonces*
   $S_p = S_p \bigcup q$
   sino si $(q \prec p)$ *entonces*
   $n_p = n_p + 1$
   si $(n_p = 0)$ *entonces*
   $p_{rank} = 1$
   $V_1 = V_1 \bigcup \{p\}$
2. $i = 1$
3. *repetir-mientras* $(V_i \neq \emptyset)$
   $Q = \emptyset$
   para cada $p \in V_i$
   para cada $q \in S_p$
   $n_q = n_q - 1$
   si $(n_q = 0)$ *entonces*
   $q_{rank} = i + 1$
   $Q = Q \bigcup \{q\}$
   $i = i + 1$
   $V_i = Q$

$i \prec_n j$ (*crowded-comparison operator*)

- *si* $(i_{rank} < j_{rank})$
- $(i_{rank} = j_{rank})$ *and* $(i_{dist} > j_{dist})$

Figura B.6: Estructura del algoritmo NSGA-II [85]

Los pasos 5 y 6 del bucle principal se utilizan cuando no se pueden añadir todos los individuos del i-ésimo frente de Pareto a la nueva población (P_{t+1}), ya que excedería el tamaño de la población (N). Todos los individuos de V_i se ordenan en orden descendente utilizando el operador crowded-comparison ($\prec_n$). Este operador se usa en todos los procesos de selección del algoritmo (reducción de la población y selección por torneo), así es necesario calcular la distancia de multitud para todos los individuos P_{t+1} (paso 4a), y no sólo para los individuos de V_i (paso 5). Como se detalla en la Figura B.6, la solución i tiene mejor puesto que la solución j si pertenece a un frente de Pareto de menor orden $i_{rank} < j_{rank}$), o si el frente de Pareto es el mismo pero la distancia de i es mayor que la de j ($i_{distance} > j_{distance}$).

Finalmente (paso 7), y usando la población P_{t+1}, los individuos se seleccionan (selección por torneo usando $\prec_n$), se cruzan y mutan para genera la nueva población Q_{t+1}.

B.5.1. Codificación del plan de trabajo

La selección de una buena representación es fundamental para resolver problemas de búsqueda. En este trabajo, se propone una representación compuesta del cromosoma.

Parte de la codificación se usa para reducir el problema de JSSP flexible a JSSP clásico y la parte restante para representar las reglas de asignación para construir el plan de trabajo. Específicamente, la asignación de máquinas se ha resuelto mediante una *codificación de trabajo en paralelo* [177]. La Figura B.7 representa esta codificación para un problema de planificación de cuatro trabajos y cuatro máquinas. Cada una de las filas de la matriz es una secuencia ordenada de operaciones $O_{i,j}$, $i = 1, \ldots, 4$, $j = 1, \ldots, 3$ (j es el índice de la operación). Cada elemento de la fila contiene dos términos: *(i)* la máquina M_k, $k = 1, \ldots, 4$, que realiza la operación y *(ii)* el tiempo de inicio $t_{k,i,j}$ de la operación $O_{i,j}$ en la máquina M_k. Por ejemplo, la operación O_1 del trabajo J_2 se realiza en la máquina M_1 en el tiempo 2.

	O_1	O_2	O_3
J_1	$(M_1, 0)$	$(M_1, 7)$	$(M_2, 2)$
J_2	$(M_1, 2)$	$(M_2, 0)$	$(M_3, 4)$
J_3	$(M_3, 0)$	$(M_1, 4)$	$(M_4, 7)$
J_4	$(M_4, 0)$	$(M_2, 4)$	$(M_3, 7)$

Figura B.7: Codificación de trabajo en paralelo de un ejemplo de planificación 4×4 (Primera parte del cromosoma)

ϕ_1	ϕ_2	ϕ_3	ϕ_4	...
0	0	3	4	...

Figura B.8: Codificación de las reglas de asignación para el ejemplo de planificación 4×4 (Segunda parte del cromosoma)

Esta codificación produce directamente planes viables. Contiene las máquinas que van a realizar cada operación y el tiempo en el que dicha operación se va a realizar. El tiempo es cero cuando la máquina no puede realizar la operación y el gen estará en blanco cuando no está definida para un determinado trabajo. De esta forma, se puede crear fácilmente un plan de trabajo conociendo las relaciones entre las operaciones y los tiempos de procesado. La Figura B.9 complementa el ejemplo de planificación 4×4 y aporta *(i)* la fechas límite de cada trabajo J_i, *(ii)* el orden de cada operación dentro del trabajo y *(iii)* el tiempo (suma de los tiempos de configuración y procesado) de cada operación $O_{i,j}$ en cada una de las máquinas de la planta de fabricación. Por ejemplo, el trabajo J_2 tiene un tiempo límite de 7 unidades de tiempo y las operaciones tienen el siguiente orden: $O_{2,2}, O_{2,1}, O_{2,3}$. El diagrama Gantt representado en la Figura B.1 muestra el plan de trabajo que se puede inferir a partir del cromosoma representado en la Figura B.7 y de los datos de la Figura B.9.

La Figura B.8 representa las reglas de asignación asociadas al problema de planificación de 4 trabajos y 4 máquinas. En esta codificación el gen i representa la regla de asignación que debe aplicarse a la operación i-ésima del plan de trabajo. A partir de estas asignaciones, una versión modificada del algoritmo Giffler and Thompson [115] construye el plan de trabajo. El código y las reglas de asignación consideradas [204] están descritas en la Tabla B.1. Por ejemplo, los primeros dos trabajos priorizarán las operaciones con el tiempo de procesado más corto (SOT).

	d_i	Order	$O_{i,j}$	M_1	M_2	M_3	M_4
J_1	9	1,3,2	$O_{1,1}$	1	4	6	9
			$O_{1,2}$	3	2	5	1
			$O_{1,3}$	4	2	1	3
J_2	7	2,1,3	$O_{2,1}$	2	8	7	1
			$O_{2,2}$	2	2	4	5
			$O_{2,3}$	6	11	2	7
J_3	8	1,2,3	$O_{3,1}$	8	5	4	9
			$O_{3,2}$	3	3	6	1
			$O_{3,3}$	7	1	8	1
J_4	11	1,2,3	$O_{4,1}$	5	10	6	4
			$O_{4,2}$	4	2	3	8
			$O_{4,3}$	7	3	4	1

Figura B.9: Fechas límite, ordenación de las operaciones y tiempos de procesado en las máquinas

Tabla B.1: Reglas de asignación de operaciones

ID	**Regla**	**Se selecciona la operación...**
0	SOT	Con el menor tiempo de procesado en la máquina
1	LOT	Con el mayor tiempo de procesado en la máquina
2	SPT	Cuyo trabajo tenga el menor tiempo total de procesado
3	LPT	Cuyo trabajo tenga el mayor tiempo total de procesado
4	SRO	Donde el número de operaciones restantes de un trabjo se menor
5	LRO	Donde el número de operaciones restantes de un trabjo se mayor
6	SRT	Donde la suma de tiempos de las operaciones restantes de un trabajo sea menor
7	LRT	Donde la suma de tiempos de las operaciones restantes de un trabajo sea mayor

B.5.2. Generación del plan

Los planes de trabajo se generan mediante la versión modificada del método *Giffer-Thompson* [115]. El pseudocódigo de este algoritmo está representado en la Figura B.10. Este algoritmo se encarga de generar los planes de trabajo a partir de la codificación del cromosoma previamente descrita. Primero, el algoritmo inserta en A todas las operaciones que están listas para ser planificadas, es decir, las primeras operaciones de cada trabajo. A puede contener varias operaciones de un mismo trabajo, ya que la relación de precedencia entre operaciones es un grafo acíclico. En cada iteración del bucle, se toma la operación $O_{i,j}$ en A con el menor tiempo potencial de finalización, se selecciona la operación $O_{x,y}$ del conjunto de operaciones a procesar por parte de la máquina asignada a $O_{i,j}$ y se elimina dicha operación del conjunto A y se añaden las operaciones que la suceden en el mismo conjunto A

La Figura B.11 representa los primeros pasos del método para los cromosomas representados en las Figuras B.7 y B.8 para la tabla de tiempos de la Figura B.9. En esta figura, un plan de trabajo se describe como un grafo dirigido $G = (V, P \cup T)$ en el cual cada nodo en V representa una operación y cada arco en $P \cup T$ representa una

1. asignar $S = \emptyset$
2. asignar $A = \{O_{i,j} | 1 \leq i \leq n, first(J_i) = j\}$
3. asignar $i = 1$
4. repetir-mientras (A is not $\emptyset$)
    a) encontrar las operaciones $O_{i,j} \in A$ con el menor tiempo de compleción potencial σ (si dos o más operaciones están igualadas, seleccionar una de ellas aleatoriamente)
    b) asignar M_k a la máquina que va a procesar la operación o
    c) asignar Q al conjunto de operaciones de A que se procesarán en M_k con un tiempo de inicio potencial menor que σ
    d) seleccionar una operación $O_{x,y} = \phi_i(Q)$ de Q, donde ϕ_i es la regla de asignación asignada a la iteración i (si dos o más operaciones están igualadas, seleccionar una de ellas aleatoriamente)
    e) eliminar $O_{x,y}$ de A
    f) añadir $O_{x,y}$ a S con el tiempo de inicio $h(O_{x,y})$
    g) añadir a A todas las operaciones del trabajo que son sucesoras de $O_{x,y}$
    h) asignar $i = i + 1$

Figura B.10: Método Giffer-Thompson modificado que se emplea para la generación del plan de trabajo. El algoritmo está modificado para usar la correspondiente relga de asignación de operaciones en cada iteración.

relación entre las operaciones. Específicamente, los arcos sin puntear representan al conjunto P de *relaciones de precedencia* mientras que los arcos punteados representan al conjunto T de relaciones tecnológicas (máquinas). Cada trabajo J_i del cromosoma representa a una fila del grafo y cada nodo está marcado con el número de la máquina que procesará a la operación. Además, cada fila del grafo puede tener ramas paralelas, ya que varias operaciones de un mismo trabajo pueden procesarse en paralelo. El conjunto A, S y Q también se representan en esta figura. Por ejemplo, las operaciones en A se marcan con círculos punteados, las operaciones en Q se marcan con cuadrados y las operaciones ya planificadas se colorean en gris.

Después del paso inicial, el conjunto A contiene a las operaciones iniciales de cada trabajo $\{O_{1,1}, O_{2,2}, O_{3,1}, O_{4,1}\}$ y Q la operación $O_{1,1}$ que tiene el menor tiempo potencial de finalización de todas las máquinas. De acuerdo con la estrategia de asignación SOT asociada a la primera iteración, únicamente la operación $O_{1,1}$ puede ser seleccionada como la operación con el menos tiempo de procesamiento para la máquina M_1. Esta operación se planifica, se elimina del conjunto A y las operaciones sucesoras en J_1 se añaden a A. En un segundo paso, $A = \{O_{1,3}, O_{2,2}, O_{3,1}, O_{4,1}\}$, se selecciona $O_{2,2}$ como la operación o con el menor tiempo de finalización, y se buscan las operaciones procesadas en la misma máquina M_2 con la estrategia SOT. Así, la segunda iteración del método define al conjunto $Q = \{O_{1,3}, O_{2,2}\}$ y selecciona aleatoriamente a la operación $O_{1,3}$. El método sigue hasta que todas las operaciones en A han sido planificadas (iteración 13).

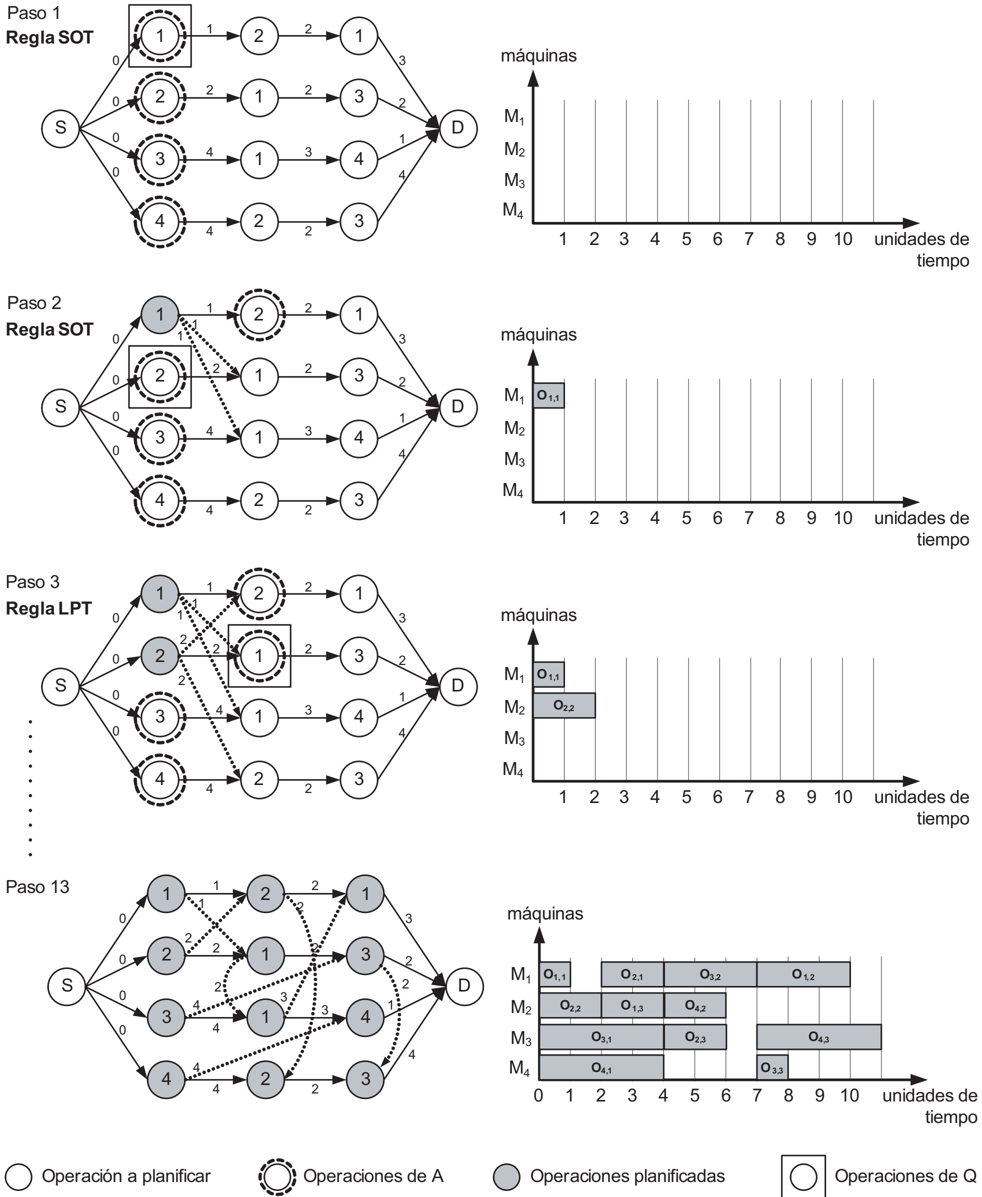


Figura B.11: Ejemplo de algunos pasos de la ejecución del método modificado de Giffer-Thompson. Cada iteración del método se representa como un diagrama de Gantt.

B.5.3. Función de cruce

En cada iteración del algoritmo evolutivo, sólo se aplica el cruce a una de las partes del cromosoma y un número aleatorio decide la parte sobre la cual realizar la operación.

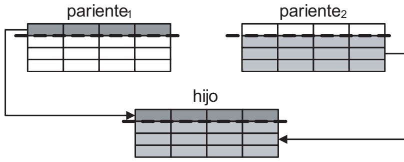


Figura B.12: Cruce de filas

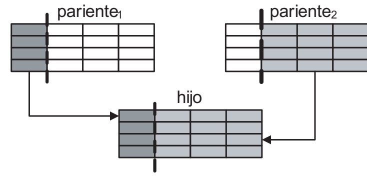


Figura B.13: Cruce de columnas

Existen dos operadores de cruce para la primera parte del cromosoma (asignación de máquinas). Ambos operadores cruzan a dos cromosomas seleccionados aleatoriamente y generan un nuevo plan de trabajo. Sin embargo, el primero operador cruza filas y afecta a los trabajos mientras que el segundo cruza columnas y, por lo tanto, afecta a las operaciones:

- Cruce de filas: como se aprecia en la Figura B.12, el hijo consiste en los $1, \ldots, k$ trabajos del $pariente_1$ y en los $k+1, \ldots, n$ trabajos del $pariente_2$.
- Cruce de columnas: como se aprecia en la Figura B.13, el hijo consiste en las $1, \ldots, k$ operaciones del $pariente_1$ y en las $k+1, \ldots, n$ operaciones del $pariente_2$.

donde k es un número aleatorio que indica el punto de cruce.

A la segunda parte del cromosoma (reglas de asignación de operaciones) se le aplica un único operador de cruce en dos puntos.

B.5.4. Función de mutación

Este operador utiliza la misma estrategia que el cruce para la selección de qué parte del cromosoma mutar. Para la asignación de máquinas, únicamente se mutan las máquinas, ya que la codificación de trabajos en paralelo no permite intercambiar trabajos u operadores en el cromosoma. Teniendo esta restricción en cuenta, se han definido los dos siguentes operadores de mutación, aunque únicamente uno de ellos se aplicará por mutación:

- Mutación aleatoria: se selecciona aleatoriamente una máquina distinta para realizar la operación $O_{i,j}$.
- Mutación con balanceo de carga: se selecciona una máquina distinta para realizar la operación $O_{i,j}$. La selección se basa en la carga de trabajo de la máquina en el plan de fabricación de forma que se busca repartir la carga de trabajo entre todas las máquinas.

donde la i-ésima fila y la j-ésima columna se seleccionan aleatoriamente. Es importante remarcar que ambos operadores de mutación generan siempre planes de trabajo viables y se evita un posible ajuste posterior a la mutación.

Un operador de mutación uniforme se aplica a las reglas de asignación de operaciones.

B.5.5. Funciones objetivo

Como se comentó anteriormente, existen muchas medidas de rendimiento de un plan de trabajo aplicables a un JSSP: *makespan*, *total flow-time*, *total lateness*, *total tardiness*, etc. En esta aplicación se optó por minimizar dos objetivos:

- C_{max}: *Makespan*. Esta medida devuelve el tiempo de finalización máximo de todos los trabajos:
$$C_{max} = max\,(C_i)$$
donde $i \in \{1, \ldots, n\}$ y el tiempo de finalización del trabajo i es:
$$C_i = max\,\{s_{k,i,j} + p_{k,i,j}\}$$
para $j \in \{1, \ldots, o\}$, y $k \in \{1, \ldots, m\}$.

- T_Σ: *Total tardiness*. Esta medida mide cuánto después de la fecha límite finaliza el trabajo. Si el trabajo finaliza antes del tiempo límite d_i, se le asigna un valor negativo:
$$T_\Sigma = \sum_{i=1}^{n} max\,(0, C_i - d_i)$$

Las medidas *makespan* y *total tardiness* son las dos medidas más importantes en el contexto de fabricación. El coste también es relevante pero se puede considerar como una variable dependiente del tiempo al igual que otros posibles objetivos. Para el ejemplo representado en la Figura B.1, el *makespan* es $C_{max} = max\,\{10, 6, 8, 11\} = 8$ y el *total tardiness* $T_\Sigma = 1{+}0{+}0{+}0 = 1$. Es necesario remarcar que ambas medidas son dependientes del tiempo de procesado de las máquinas y que esta estimación tiene una gran importancia en cualquier industria [131] y más en este tipo de industria donde los tiempos dependen de muchos factores.

B.6. Resultados de la planificación

El método de planificación ha sido validado con un conjunto de problemas *benchmark* del clásico JSSP. Específicamente, se usó un conjunto de casos de prueba creados por Hurink et al. [142] que adaptan los problemas del JSSP a problemas de JSSP flexible. En estos problemas, cada operación tienen asignada un conjunto de máquinas y no únicamente una máquina M_k como en el JSSP clásico. Los datos de los casos de prueba y las mejores soluciones encontradas hasta la fecha pueden consultarse en [175]. Las fechas límite de los casos de prueba han sido definidas acorde a [165]: los trabajos 2, 3 y 11 tienen una fecha límite 1,5 veces mayor a su tiempo de procesado; los trabajos n, donde n es el número de trabajos del problema (JSSP $n \times m$), tienen

una fecha límite igual a su tiempo de procesado; y los demás trabajos tienen una fecha límite igual a dos veces su tiempo de procesado.

El principal objetivo de esta validación ha sido medir el rendimiento de la aproximación NSGA-II por la que se optó para resolver la tarea *proponer planes*. Por ello, se han añadido otros métodos que resuelven dicha tarea en el marco de conocimiento propuesto y se han analizado sus resultados. Al igual que el NSGA-II, estos métodos se resuelven mediante tareas primitivas que representan otros MOEAs:

- FastPGA [101]: el Fast Pareto Genetic Algorithm utiliza una nueva función de asignación de bondad y una nueva estrategia de clasificación donde la evaluación de cada solución puede llegar a ser computacionalmente costosa. Esto suele ocurrir cuando existen restricciones de recursos o tiempo a la hora de encontrar una solución. Este algoritmo también introduce un operador de regulación de la población para adaptar dinámicamente el tamaño de la población por encima de las necesidades especificadas por el usuario.

- GDE3 [160]: el Generalized Differential Evolution 3 es una extensión del algoritmo Differential Evolution (DE) para la optimización global con un número arbitrario de objetivos y restricciones. GDE3 mejora versiones anteriores del GDE para problemas multi-objetivos consiguiendo una mejor distribución de las soluciones.

- PAES [154]: el Pareto Archived Evolution Strategy consiste en una estrategia de evolución 1+1 (un único pariente genera un único descendiente) en combinación con el archivo histórico que registra las soluciones no dominadas encontradas anteriormente. Este archivo se utiliza como un conjunto de referencias contra las que se compara cada individuo que ha sufrido una mutación. Este archivo histórico es el mecanismo elitista adoptado por el algoritmo PAES.

- PESA-II [79]: es una versión mejorada del PAES donde la unidad de selección es una hipercaja del espacio de objetivos. Esta técnica, en lugar de asignar una bondad parcial a un individuo, asigna la bondad a una hipercaja del espacio de objetivos que siempre contiene al menos uno de los individuos del frente actual del Pareto. De esta forma se obtiene siempre una hipercaja y el individuo seleccionado se toma aleatotoriamente de la hipercaja.

- SPEA-II [311]: el Strength Pareto Evolutionary Algorithm II asigna un valor de bondad a cada individuo que es la suma de su bondad pura y de una estimación de densidad. El algoritmo aplica operadores de selección, cruce y mutación para completar el archivo de individuos; entonces, se copian los individuos no dominados tanto de la población actual como del archivo a la nueva población. Si el número de individuos no dominados es mayor que el tamaño de la población, se aplica un operador de ajuste que se basa en el cálculo de las distancias de los k vecinos más próximos. De esta forma, se seleccionan los individuos con la menor distancia a otros individuos.

Tabla B.2: Resumen de los resultados obtenidos para los métodos multi-objetivo considerados. La primera columna de la tabla identifica el caso de prueba mientras que las columnas μ identifican un método (F=FastPGA, G=GDE3, N=NSGA-II, P=PAES, E=PESA-II, S=SPEA-II).

	μ	C_{max}		T_{Σ}		μ	C_{max}		T_{Σ}	
		mejor	media	mejor	media		mejor	media	mejor	media
1	F	852	882.0	1171	1390.0	G	855	898.0	1332	1560.5
	N	814	881.6	**937**	1432.1	P	856	914.0	1563	2031.0
	E	846	866.7	1286	1645.7	S	**806**	859.3	965	1481.7
2	F	815	863.3	1155	1734.7	G	843	888.5	1343	1477.0
	N	804	844.5	**921**	1191.0	P	829	919.5	1147	1803.0
	E	813	837.0	1254	1424.6	S	**802**	838.5	1079	1297.3
3	F	5848	6289.1	5324	9459.3	G	6657	6839.0	9203	9960.0
	N	**5552**	5712.2	**4520**	6780.7	P	6235	6235.0	6118	6118.0
	E	5725	6292.3	5943	7530.2	S	5927	6607.8	5358	7692.3
4	F	**5970**	6467.3	**299**	1486.0	G	6934	7492.2	478	1343.0
	N	6078	6760.6	299	1380.7	P	6434	6757.0	299	299.0
	E	6136	7001.0	299	299.0	S	6090	7001.3	299	499.8
5	F	898	916.5	3953	4374.9	G	921	1057.4	4484	4993.8
	N	877	953.0	3929	4471.9	P	926	1019.1	3983	4772.3
	E	904	945.1	3809	4373.9	S	**864**	981.0	**3604**	3860.1
6	F	1181	1291.7	8666	9405.5	G	1216	1379.0	9468	10230.3
	N	**1145**	1237.5	8577	10072.0	P	1264	1347.0	8688	10752.3
	E	1168	1262.9	8117	10551.8	S	1151	1206.3	**7926**	9065.6
7	F	857	907.2	55	309.3	G	960	1045.1	332	625.8
	N	**802**	848.55	**56**	103.50	P	872	1117.1	105	258.7
	E	885	937.6	108	217.0	S	861	1045.1	97	178.0
8	F	1117	1175.3	1405	1987.2	G	1194	1297.8	2040	2485.8
	N	**1104**	1192.4	**729**	1342.8	P	1194	1265.2	1867	2521.6
	E	1129	1223.0	1205	2039.3	S	1113	1223.7	776	1758.5
9	F	1427	1576.3	5254	7238.7	G	1456	1487.5	5343	6622.5
	N	**1355**	1440.6	**4955**	5802.0	P	1413	1601.6	5987	6819.3
	E	1395	1470.8	5617	6726.5	S	1379	1499.4	5109	5409.4
10	F	1955	2052.2	18793	20531.0	G	2048	2100.7	19430	22132.4
	N	**1916**	1965.0	**18453**	20126.6	P	2064	2201.0	20315	22874.2
	E	1964	2091.1	18825	20912.3	S	1938	2038.5	17814	19361.6
11	F	1996	2056.0	17846	20263.5	G	2091	2216.1	19580	21533.3
	N	**1973**	2116.8	**17555**	19718.7	P	2049	2186.6	19180	22096.4
	E	2054	2132.6	19591	21067.6	S	1987	2075.1	18329	19690.8
12	F	1469	1572.0	334	542.0	G	1480	1608.2	410	1235.6
	N	1357	1471.0	**186**	712.2	P	1479	1582.8	389	772.4
	E	1418	1501.0	363	929.6	S	**1395**	1456.0	233	598.1
13	F	1408	1518.6	243	892.8	G	1456	1560.3	686	872.6
	N	**1380**	1484.8	**101**	414.6	P	1456	1566.4	542	838.0
	E	1414	1484.6	297	667.1	S	1382	1443.4	229	567.7
14	F	**741**	924.8	42	167.2	G	916	1060.0	227	385.0
	N	770	969.0	**31**	104.7	P	840	893.1	163	503.0
	E	805	1009.6	74	151.9	S	754	893.9	55	275.6
15	F	1090	1277.0	7684	8002.6	G	1168	1211.4	8760	9949.0
	N	**1060**	1170.7	**7099**	8474.5	P	1104	1238.0	8593	10108.0
	E	1105	1168.9	8092	9552.2	S	1091	1196.1	7492	8668.0
16	F	**702**	763.3	53	316.2	G	798	869.2	211	334.0
	N	707	770.7	**27**	83.7	P	787	798.3	28	454.2
	E	709	842.4	76	141.7	S	733	779.4	37	146.5
17	F	844	902.3	123	135.2	G	901	1035.8	275	545.1
	N	823	920.1	72	331.4	P	856	862.3	116	268.3
	E	840	1001.2	97	330.1	S	**813**	894.7	**68**	152.3

La Tabla B.2[3] muestra el mejor y la media de los *makespan* y el mejor y la media de los *total tardiness* para 5 ejecuciones de cada una de las aproximaciones. Cada

[3] Los resultados se han obtenido aplicando el software jMetal [92].

Tabla B.3: Comparación entre los resultados obtenidos por B =NSGA-II y los demás métodos

Caso	Medida	A				
		FastPGA	GDE3	PAES	PESA-II	SPEA-II
1	$\tilde{C}(A,B)$	0.32	0.11	0.00	0.57	0.96
	$\tilde{C}(B,A)$	1.00	1.00	1.00	0.93	0.67
2	$\tilde{C}(A,B)$	0.00	0.00	0.35	0.04	0.62
	$\tilde{C}(B,A)$	1.00	1.00	1.00	1.00	0.86
3	$\tilde{C}(A,B)$	0.33	0.00	0.23	0.34	0.33
	$\tilde{C}(B,A)$	1.00	1.00	1.00	0.69	1.00
4	$\tilde{C}(A,B)$	0.30	0.00	0.25	0.40	0.40
	$\tilde{C}(B,A)$	0.56	1.00	0.56	0.39	0.77
5	$\tilde{C}(A,B)$	0.70	0.00	0.09	0.06	0.65
	$\tilde{C}(B,A)$	0.93	1.00	0.73	0.95	0.14
6	$\tilde{C}(A,B)$	0.49	0.00	0.02	0.53	0.96
	$\tilde{C}(B,A)$	0.88	1.00	1.00	0.82	0.43
7	$\tilde{C}(A,B)$	0.50	0.00	0.01	0.24	0.48
	$\tilde{C}(B,A)$	0.86	1.00	1.00	1.00	1.00
8	$\tilde{C}(A,B)$	0.03	0.00	0.00	0.14	0.64
	$\tilde{C}(B,A)$	1.00	1.00	1.00	1.00	0.94
9	$\tilde{C}(A,B)$	0.38	0.19	0.03	0.50	0.84
	$\tilde{C}(B,A)$	0.97	1.00	1.00	0.96	0.78
10	$\tilde{C}(A,B)$	0.63	0.10	0.00	0.43	0.77
	$\tilde{C}(B,A)$	1.00	1.00	1.00	1.00	0.91
11	$\tilde{C}(A,B)$	0.70	0.00	0.00	0.00	0.76
	$\tilde{C}(B,A)$	0.94	1.00	1.00	1.00	0.87
12	$\tilde{C}(A,B)$	0.00	0.00	0.00	0.10	0.59
	$\tilde{C}(B,A)$	1.00	1.00	1.00	1.00	0.86
13	$\tilde{C}(A,B)$	0.28	0.00	0.00	0.31	0.83
	$\tilde{C}(B,A)$	1.00	1.00	1.00	1.00	0.72
14	$\tilde{C}(A,B)$	0.92	0.00	0.00	0.33	0.89
	$\tilde{C}(B,A)$	0.72	1.00	1.00	1.00	0.73
15	$\tilde{C}(A,B)$	0.42	0.00	0.37	0.35	0.78
	$\tilde{C}(B,A)$	0.86	1.00	0.93	0.93	0.76
16	$\tilde{C}(A,B)$	0.36	0.00	0.02	0.17	0.29
	$\tilde{C}(B,A)$	0.82	1.00	1.00	0.96	0.99
17	$\tilde{C}(A,B)$	0.62	0.00	0.73	0.64	0.89
	$\tilde{C}(B,A)$	0.82	1.00	0.82	0.93	0.72

ejecución consistió en 25.000 iteraciones de una población de 100 individuos, con ratios de cruce y mutación iguales a 0,95 y 0,05 respectivamente. En cada caso de prueba, el mejor *makespan* y el mejor *total tardiness* de todas las aproximaciones está destacado en negrita. Para facilitar la visión de la tabla, un número en la primera columna se corresponde con la identificación de un caso de prueba. La asociación entre números y casos de prueba es la siguiente:

1. abz8 (10×10).
2. abz9 (10×10).
3. car5 (10×06).
4. car6 (08×09).
5. la06 (15×05).
6. la11 (20×05).
7. la16 (10×10).
8. la24 (15×10).
9. la29 (20×10).

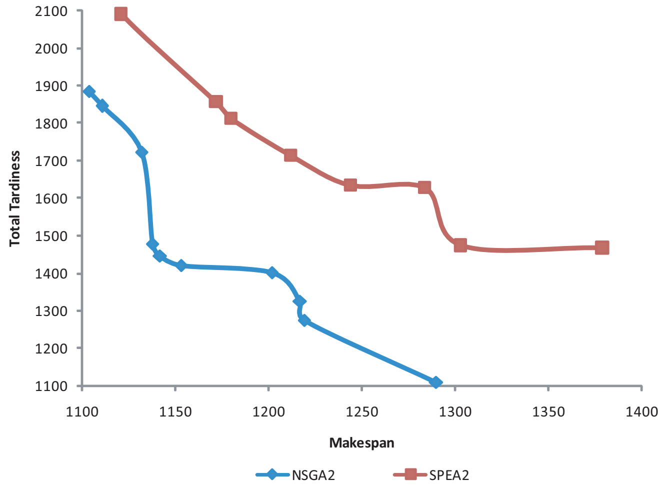


Figura B.14: Frente Pareto final del NSGA-II y del SPEA-II para el problema *la24*

10. la34 (30×10).
11. la35 (30×10).
12. la39 (15×15).
13. la40 (15×15).
14. mt10 (10×10).
15. mt20 (20×05).
16. orb8 (10×10).
17. orb9 (10×10).

Como se muestra en la Tabla B.2, el NSGA-II obtiene el mejor *makespan* en 9 de los 17 problemas. La Tabla B.3 resume los resultados anteriores con una comparación entre el NSGA-II y los algoritmos FastPGA, GDE3, PAES, PESA-II y SPEA-II. La métrica $\tilde{\mathcal{C}}$ [165] se ha usado para comparar la aproximación óptima del Pareto entre el NSGA-II y el resto de los MOEAs. Específicamente, $\tilde{\mathcal{C}}(A, B)$ mide la fracción de miembros del conjunto B que son dominados por miembros del conjunto A:

$$\tilde{\mathcal{C}}(A, B) = \frac{|\,\{b \in B : \exists a \in A, a \succ b\}\,|}{|B|}$$

En la comparativa de la Tabla B.3 se aprecia como el NSGA-II es mejor que los demás MOEAs en la mayor parte de las situaciones. NSGA-II obtiene una menor medida de $\tilde{\mathcal{C}}$ que FastPGA en 16 problemas, que GDE3 y PAES en todos los problemas, que PESA-II en 16 problemas, y que SPEA-II en 9 de los 17 problemas. Es importante remarcar que el SPEA-II, el algoritmo que obtiene la segundo mejor *bondad*, únicamente mejora claramente al NSGA-II en los casos *abz8*, *la06*, *la11*, *mt10* y *orb9*. En los demás casos, los resultados de ambas aproximaciones son similares. Para ilustrar la convergencia y diversidad de las soluciones, la Figura B.14 representa a las soluciones no dominadas de la última generación producidas para el caso de prueba *la24* por el NSGA-II y el SPEA-II. Se puede apreciar como las soluciones de ambos algoritmos están bien esparcidas y convergen. Sin embargo, el NSGA-II produce más

soluciones no dominadas para la mayoría de los problemas. Para más información acerca del algoritmo evolutivo o de los resultados consultar [282].

C

Correspondencias entre OWL y FLORA-2

Este capítulo muestra las correspondencias definidas entre OWL y FLORA-2 F-Logic. Estas correspondencias se emplearon para traducir las ontologías OWL que anotan los servicios OWL-S a su correspondiente versión en FLORA-2. Esta traducción se basa en varios trabajos [107, 185] que preservan la semántica de la mayor parte de los axiomas definidos en la ABox y TBox de OWL. Otros axiomas, como por ejemplo las restricciones de cardinalidad, se incorporan como reglas de inferencia del mismo modo que se definieron la mayor parte de los axiomas a lo largo de este trabajo. La siguente tabla C.1 recoge las principales correspondencias definidas en este trabajo.

Tabla C.1: Correspondencias entre OWL y FLORA-2

Ejemplo en OWL	Traducción a FLORA-2
subClassOf	
<owl:Class rdf:ID="Wenge"> <rdfs:subClassOf rdf:resource="&material;Madera"/> </owl:Class>	Wenge::'&material;Madera'
equivalentClass	
<owl:Class rdf:ID="Union"> <owl:equivalentClass rdf:resource="Juntura"/> </owl:Class>	Equivalencia simbólica: Union :=: Juntura Equivalencia subclases: ?x::Union :- ?x::Juntura ?x::Juntura :- ?x::Union Equivalencia instancias: ?x:Union :- ?x:Juntura ?x:Juntura :- ?x:Union
unionOf	

<table>
<tr><td><owl:Class rdf:ID="Madera">
<owl:unionOf rdf:parseType=
"Collection">
<owl:Class rdf:about=
"#MaderaMaciza"/>
<owl:Class rdf:about=
"#Aglomerado"/>
</owl:unionOf>
</owl:Class></td><td>Madera :=:
(Maciza ; Aglomerado)</td></tr>
<tr><td colspan="2">intersectionOf</td></tr>
<tr><td><owl:Class rdf:ID="MDFHidrofugo">
<owl:intersectionOf
rdf:parseType="Collection">
<owl:Class rdf:about="#MDF"/>
<owl:Class rdf:about="#Hidrofugo"/>
</owl:intersectionOf>
</owl:Class></td><td>MDFHidrofugo :=:
(MDF, Hidrofugo)</td></tr>
<tr><td colspan="2">complementOf</td></tr>
<tr><td><owl:Class rdf:ID="NoHidrofugo">
<owl:complementOf
rdf:resource="#Hidrofugo"/>
</owl:Class></td><td>Thing es la raíz de la jerarquía:
NoHidrofugo :=:
(Thing - Hidrofugo)</td></tr>
<tr><td colspan="2">disjointWith</td></tr>
<tr><td><owl:Class rdf:ID="Madera">
<owl:disjointWith
rdf:resource="#Metal"/>
</owl:Class></td><td>disjoint(Madera,Metal).
%error(disjoint, ?_c1, ?_c2) :-
disjoint(?_c1,?_c2),
?_x:?_c1,
?_x:?_c2.</td></tr>
<tr><td colspan="2">oneOf</td></tr>
<tr><td><owl:Class rdf:ID="Acabado">
<owl:oneOf rdf:parseType="Collection">
<owl:Thing rdf:about="#Brillante"/>
<owl:Thing rdf:about="#Mate"/>
<owl:Thing rdf:about="#Satinado"/>
</owl:oneOf>
</owl:Class></td><td>Brillante:Acabado.
Mate:Acabado.
Satinado:Acabado.
oneOf(Acabado,
[Brillante,Mate,Satinado]).
%error(oneOf, ?_x, ?_c) :-
?_x:?_c,
oneOf(?_c,?_l),
not(member(?_x,?_l)).</td></tr>
<tr><td colspan="2">allValuesFrom</td></tr>
</table>

```
Todos los muebles se venden en
la carpintería:

<owl:Class rdf:ID="Mueble">
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty
    rdf:resource="#puntoDeVenta"/>
   <owl:allValuesFrom
    rdf:resource="#Carpinteria"/>
  </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
```

```
Mueble[
  puntoDeVenta *=> Carpinteria
].

?_y:Carpinteria :-
  ?_x:Mueble,
  ?_x:[puntoDeVenta -> ?_y].
```

someValuesFrom

```
Uno de los lugares en los que se
vende un mueble es en la carpintería:

<owl:Class rdf:ID="Mueble">
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty
    rdf:resource="#puntoDeVenta"/>
   <owl:someValuesFrom
    rdf:resource="#Carpinteria"/>
   </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

```
someValuesFrom(Mueble,
 puntoDeVenta,Carpinteria).

%error(someValues, ?_x, ?_c) :-
 someValuesFrom(?_c,?_p,?_r),
 ?_x:?_c,
 not ?_x.?_p : ?_r.
```

hasValue

```
<owl:Class rdf:ID="MaderaPesada">
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty
    rdf:resource="#tieneDensidad"/>
   <owl:hasValue
    rdf:resource="#AltaDensidad"/>
  </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
```

```
hasValue(MaderaPesada,
 tieneDensidad,AltaDensidad).

%error(hasValue, ?_x, ?_c) :-
 hasValue(?_c,?_p,?_v),
 ?_x:?_c,
 not(?_x[?_p -> ?_v]).
```

maxCardinality

Un tablero tiene que procesar a
lo sumo 4 cantos:
 	
<owl:Class rdf:ID="Tablero">
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty
    rdf:resource="#tieneCanto"/>
   <owl:maxCardinality rdf:datatype=
    "&xsd;nonNegativeInteger">4
   </owl:maxCardinality>
  </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
	Tablero[
	 tieneCantos{0:5} *=> Canto
	].
	
	%error(maxCardinality, ?_x, ?_c) :-
	 Cardinality[%_check(
	  ?_x[tieneCanto=>?])]
	   @_typecheck.

minCardinality	

Una mesa tiene al menos una pata:
 	
<owl:Class rdf:ID="Mesa">
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty
    rdf:resource="#tienePata"/>
   <owl:minCardinality rdf:datatype=
    "&xsd;nonNegativeInteger">1
   </owl:minCardinality>
  </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
	Mesa[
	 tienePata{1:*} *=> Pieza
	].
	
	%error(minCardinality, ?_x, ?_c) :-
	 Cardinality[%_check(
	  ?_x[tieneCanto=>?])]
	   @_typecheck.

cardinality	

Un tablero tiene 2 caras:
<owl:Class rdf:ID="Tablero">
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty
    rdf:resource="#tieneCaras"/>
   <owl:cardinality rdf:datatype=
    "&xsd;nonNegativeInteger">2
   </owl:cardinality>
  </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
	Tablero[
	 tieneCaras{1:*} *=> Cara
	].
	
	%error(cardinality, ?_x, ?_c) :-
	 Cardinality[%_check(
	  ?_x[tieneCanto=>?])]
	   @_typecheck.

domain/range	
<owl:ObjectProperty rdf:ID="tieneCanto"> <rdfs:domain rdf:resource="#Tablero"/> <rdfs:range rdf:resource="#Canto"/> </owl:ObjectProperty>	Tablero[tieneCanto *=> Canto].
subPropertyOf	

<table>
<tr><td><owl:ObjectProperty
rdf:ID="tieneUnion">
<rdfs:subPropertyOf
rdf:resource="#tieneUnionMilano"/>
</owl:ObjectProperty></td><td>?_x[tieneUnionMilano -> ?_y] :-
?_x[tieneUnion -> ?_y].</td></tr>
<tr><td colspan="2">equivalentProperty</td></tr>
<tr><td><owl:ObjectProperty
rdf:about="#tieneMaterial">
<owl:equivalentProperty
rdf:resource="compuestaDe"/>
</owl:ObjectProperty></td><td>?_x[tieneMaterial ->?_y] :-
?_x[compuestaDe ->?_y].
?_x[compuestaDe ->?_y] :-
?_x[tieneMaterial ->?_y].</td></tr>
<tr><td colspan="2">inverseOf</td></tr>
<tr><td><owl:ObjectProperty rdf:ID="vende">
<owl:inverseOf
rdf:resource="#puntoDeVenta" />
</owl:ObjectProperty></td><td>?_x[vende -> ?_y] :-
?_y[puntoDeVenta ->?_x].

?_x[puntoDeVenta -> ?_y]:-
?_y[vende ->?_x].</td></tr>
<tr><td colspan="2">FunctionalProperty</td></tr>
<tr><td><owl:ObjectProperty
rdf:ID="fechaFabricacion">
<rdf:type rdf:resource=
"&owl;FunctionalProperty"/>
</owl:ObjectProperty></td><td>_object[
fechaFabricacion{1:1} *=>
_object].</td></tr>
<tr><td colspan="2">InverseFunctionalProperty</td></tr>
<tr><td><owl:ObjectProperty rdf:ID="vende">
<rdf:type rdf:resource=
"&owl;InverseFunctionalProperty"/>
<owl:inverseOf
rdf:resource="#puntoDeVenta"/>
</owl:ObjectProperty></td><td>?_x[vende ->?_y] :-
?_y[puntoDeVenta ->?_x].

?_x[puntoDeVenta ->?_y] :-
?_y[vende ->?_x].

_object[vende{1:1} *=> _object].</td></tr>
<tr><td colspan="2">TransitiveProperty</td></tr>
<tr><td><owl:ObjectProperty
rdf:ID="tienePieza">
<rdf:type rdf:resource=
"&owl;TransitiveProperty" />
</owl:ObjectProperty></td><td>TransitiveProperty(tienePieza).

?_x[?_p -> ?_z]:-
TransitiveProperty(?_p),
?_x[?_p -> ?_y],
?_y[?_p -> ?_z].</td></tr>
<tr><td colspan="2">SymmetricProperty</td></tr>
<tr><td><owl:ObjectProperty
rdf:ID="piezaAdyacente">
<rdf:type rdf:resource=
"&owl;SymmetricProperty" />
</owl:ObjectProperty></td><td>SymmetricProperty(piezaAdyacente).

?_x[?_p -> ?_y]:-
SymmetricProperty(?_p),
?_y[?_p -> ?_x].</td></tr>
</table>

Bibliografía

[1] A. Aamodt. *A Knowledge-Intensive, Integrated Approach to Problem Solving and Sustained Learning.* PhD thesis, Universitetet I Trondheim, Trondheim, Norway, 1992.

[2] E. H. L. Aarts, P. J. M. van Laarhoven, J. K. Lenstra, and N. L. J. Ulder. A Computational Study of Local Search Algorithms for Job Shop Scheduling. *ORSA Journal on Computing*, 6(2):118–125, 1994.

[3] J. Adams, E. Balas, and D. Zawack. The Shifting Bottleneck Procedure for Job Shop Scheduling. *Management Science*, 34(3):391–401, 1988.

[4] A. Agrawal, M. Amend, M. Das, M. Ford, C. Keller, M. Kloppmann, D. König, F. Leymann, R. Müller, G. Pfau, K. Plösser, R. Rangaswamy, A. Rickayzen, M. Rowley, P. Schmidt, I. Trickovic, A. Yiu, and M. Zeller. WS-BPEL Extension for People (BPEL4People), Version 1.0. Technical report, OASIS Web Services Business Process Execution Language (WSBPEL) TC, June 2007.

[5] H. Akkermans, B. J. Wielinga, and G. Schreiber. Steps in Constructing Problem Solving Methods. In *Proceedings of the 7th European Workshop on Knowledge Acquisition for Knowledge-Based Systems (EKAW'93)*, pages 45–65, London, UK, Sept. 1993. Springer-Verlag.

[6] R. Alcalá, J. Alcalá Fdez, J. Casillas, O. Cordón, and F. Herrera. Local Identification of Prototypes for Genetic Learning of Accurate TSK Fuzzy Rule-Based Systems: Research Articles. *Int. J. Intell. Syst.*, 22(9):909–941, 2007.

[7] J. Alcalá Fdez, L. Sánchez, S. García, M. J. del Jesus, S. Ventura, J. M. Garrell, J. Otero, C. Romero, J. Bacardit, V. M. Rivas, J. C. Fernández, and F. Herrera. KEEL: A Software Tool to Assess Evolutionary Algorithms for Data Mining Problems. *Soft Comput.*, 13(3):307–318, 2008.

[8] A. Allahverdi, T. C. E. Ng, C. T. Cheng, and M. Y. Kovalyov. A Survey of Scheduling Problems with Setup Times or Costs. *European Journal of Operational Research*, 187(3):985–1032, 2006.

[9] A. Alves, A. Arkin, S. Askary, C. Barreto, Ben, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C. K. Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS Web Services Business Process Execution Language (WSBPEL) TC, Apr. 2007.

[10] S. W. Ambler. *The Object Primer : Agile Model-Driven Development with UML 2.0*. Cambridge University Press, Mar. 2004.

[11] V. Ambriola, R. Conradi, and A. Fuggetta. Assessing Process-Centered Software Engineering Environments. *ACM Trans. Softw. Eng. Methodol.*, 6(3):283–328, 1997.

[12] R. Amorim, M. Lama, E. Sánchez, A. Riera, and X. A. Vila. A Learning Design Ontology Based on the IMS Specification. *Journal of Educational Technology and Society*, 9:38–57, 2006.

[13] J. Andersson. *Multiobjective Optimization in Engineering Design*. PhD Dissertation, Linköpings University, Linköping, Sweden, 2001.

[14] E. Andonoff, L. Bouzguenda, and C. Hanachi. Specifying Workflow Web Services Using Petri Nets with Objects and Generating of Their OWL-S Specifications. In *E-Commerce and Web Technologies*, volume 3590 of *Lecture Notes in Computer Science*, pages 41–52. Springer Berlin / Heidelberg, Aug. 2005.

[15] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services version 1.1. Technical report, OASIS Web Services Business Process Execution Language (WSBPEL) TC, May 2003. OASIS standard.

[16] A. Anjewierden and G. Schreiber. CML2 Syntax (2.2.1). Technical report, SWI, University of Amsterdam, aug 2003.

[17] Antoon Goderis, Ulrike Sattler, Phillip Lord, and Carole Goble. Seven Bottlenecks to Workflow Reuse and Repurposing. In *4th International Semantic Web Conference (ISWC 2005)*, volume 3729, pages 323–337, Galway, Ireland, Nov. 2005. Springer Berlin / Heidelberg.

[18] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, S. Riemer, S. Struble, P. Takacsi Nagy, I. Trickovic, and S. Zimek. *Web Service Choreography Interface (WSCI) 1.0*. World Wide Web Consortium (W3C), Aug. 2002. W3C submission.

[19] P. Armenise, S. Bandinelli, C. Ghezzi, and A. Morzenti. A Survey and Assessment of Software Process Representation Formalisms. *Int. Journal on Software Engineering and Knowledge Engineering*, 3(3):401–426, 1993.

[20] S. Arroyo and J. M. López Cobo. Describing Web Services with Semantic Metadata. *International Journal of Metadata, Semantics and Ontologies*, 1(1):76–82, 2006.

[21] E. Balas, N. Simonetti, and A. Vazacopoulos. Job Shop Scheduling with Setup Times, Deadlines and Precedence Constraints. In *Proceedings of the 2nd Multidisciplinary International Conference on Scheduling: Theory and Application (MISTA 2005)*, pages 520–532, New York, USA, July 2005.

[22] M. Ballicu, A. Giua, and C. Seatzu. Job-Shop Scheduling Models with Set-Up Times. In *Proceedings of the 2002 IEEE International Conference on Systems, Man and Cybernetics*, pages 95–100, Hammamet, Tunisia, Oct. 2002. IEEE Service Center.

[23] H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics.* North Holland, 1981.

[24] S. Beco, B. Cantalupo, N. Matskanis, and M. Surridge. Putting semantics in grid workflow management: the owl-ws approach. Technical Report NextGrid Deliverable, University of Southampton IT Innovation Centre, 2006.

[25] S. Beco, B. Cantalupo, N. Matskanis, and M. Surridge. Putting Semantics in Grid Workflow Management: the OWL-WS Approach. In *Proceedings of the GGF16 Semantic Grid Workshop*, pages 1–5, Athens, Greece, Feb. 2006.

[26] V. R. Benjamins. On a Role of Problem Solving Methods in Knowledge Acquisition. In *A Future for Knowledge Acquisition, 8th European Knowledge Acquisition Workshop, EKAW'94*, pages 137–157, London, UK, Sept. 1994. Springer-Verlag.

[27] V. R. Benjamins and D. Fensel. Editorial: Problem-Solving Methods. *International Journal of Human-Computer Studies*, 49(4):305–313, 1998.

[28] V. R. Benjamins, E. Plaza, E. Motta, D. Fensel, R. Studer, B. Wielinga, G. Schreiber, Z. Zdrahal, and S. Decker. IBROW3 - An Intelligent Brokering Service for Knowledge-Component Reuse on the World-Wide Web. In *Proceedings of the 11th Knowledge Acquisition for Knowledge-Based System Workshop (KAW´98)*, pages 1–12, Banff, Canada, Apr. 1998.

[29] F. J. Berlanga, M. J. del Jesus, and F. Herrera. Learning Compact Fuzzy Rule-Based Classification Systems with Genetic Programming. In *Proceedings of the 4th Conference of the European Society for Fuzzy Logic and Technology (EUSFLAT)*, pages 1027–1032, Barcelona, Spain, Sept. 2005.

[30] M. Bernauer, G. Kappel, G. Kramler, and W. Retschitzegger. Specification of Interorganizational Workflows - A Comparison of Approaches. In *Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2003)*, pages 30–36, Orlando, Florida, USA, July 2003. International Institute of Informatics and Systemics.

[31] T. Berners Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[32] S. Binato, W. J. Hery, D. M. Loewenstern, and M. G. C. Resende. *Essays and Surveys on Metaheuristics*, chapter A GRASP for Job Shop Scheduling, pages 59–79. Kluwer Academic Publishers, Apr. 2001.

[33] P. V. Biron, K. Permanente, and A. Malhotra. *XML Schema Part 2: Datatypes Second Edition*. World Wide Web Consortium (W3C), Oct. 2004. W3C recommendation.

[34] F. Bonchi, A. Brogi, S. Corfini, and F. Gadducci. Compositional Specification of Web Services Via Behavioural Equivalence of Nets: A Case Study. In *Proceedings of the 29th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ATPN'08)*, pages 52–71, Berlin, Heidelberg, June 2008. Springer-Verlag.

[35] G. Boothroyd. *Assembly Automation and Product Design*. Marcel Dekker, Aug. 1991.

[36] G. Boothroyd, P. Dewhurst, and W. Knight. *Product Design for Manufacture and Assembly*. Marcel Dekker, Feb. 1994.

[37] E. Börger. Modeling Workflow Patterns from First Principles. In *Conceptual Modeling - ER 2007*, volume 4801 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin / Heidelberg, 2007.

[38] E. Börger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. In *Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST '00)*, pages 293–308, Iowa City, Iowa, USA, May 2000. Springer-Verlag.

[39] E. Börger and D. Rosenzweig. A Mathematical Definition of Full Prolog. *Science of Computer Programming*, 24(3):249–286, 1995.

[40] E. Börger and W. Schulte. *A Programmer Friendly Modular Definition of the Semantics of Java*. Springer, 1998.

[41] E. Börger and B. Thalheim. Modeling Workflows, Interaction Patterns, Web Services and Business Processes: The ASM-Based Approach. In *Proceedings of the 1st International Conference on Abstract State Machines, B and Z (ABZ '08)*, pages 24–38, London, UK, Sept. 2008. Springer-Verlag.

[42] J. Branke and D. C. Mattfeld. Anticipation in Dynamic Optimization: The Scheduling Case. In *Proceedings of the 6th International Conference on Parallel Problem Solving from Nature*, pages 253–262, London, UK, Sept. 2000. Springer-Verlag.

[43] V. d. v. Breuker. *Common KADS Library for Expertise Modelling*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 1994.

[44] C. A. Brizuela Rodríguez. *Genetic Algorithms for Shop scheduling Problems: Partial Enumeration and Stochastic Heuristics.* PhD Dissertation, Kyoto Institute of Technology, Japan, 2000.

[45] S. Brockmans, M. Ehrig, A. Koschmider, A. Oberweis, and R. Studer. Semantic Alignment of Business Processes. In *Proceedings of the Eighth International Conference on Enterprise Information Systems (ICEIS 2006)*, pages 191–196, Paphos, Cyprus, May 2006.

[46] P. Brucker, B. Jurish, and B. Sieners. A Branch and Bound Algorithm for the Job-Shop Scheduling Problem. *Discrete and Applied Mathematics*, 49(1-3):107–127, 1994.

[47] J. d. Bruijn, D. Fensel, U. Keller, M. Kifer, H. Lausen, R. Krummenacher, A. Polleres, and L. Predoiu. *Web Service Modeling Language (WSML).* World Wide Web Consortium (W3C), June 2005. W3C submission.

[48] D. Burgos, C. Tattersall, and E. J. R. Koper. Utilización de Estándares en el Aprendizaje Virtual. Funcionalidades Didácticas de la Especificación IMS Learning Design, 2002. II Jornadas Campus Virtual, Universidad Complutense, Madrid, Spain.

[49] B. Cantalupo, L. Giammarino, N. Matskanis, M. Surridge, and F. Silvestri. Semantic Workflow Representation and Samples. Technical Report P5.3.1 NextGrid Deliverable, University of Southampton IT Innovation Centre, 2005.

[50] J. Cardoso, R. P. Bostrom, and A. Sheth. Workflow Management Systems vs. ERP Systems: Differences, Commonalities, and Applications. *Information Technology and Management*, 5(3):319–338, 2004.

[51] J. Carlier and E. Pinson. An Algorithm for Solving the Job-Shop Scheduling Problem. *Management Science*, 35(2):164–176, 1989.

[52] S. Carlsen. Action Port Model: A Mixed Paradigm Conceptual Workflow Modeling Language. In *Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems (COOPIS '98)*, pages 300–309, Washington, DC, USA, Aug. 1998. IEEE Computer Society.

[53] B. Carse, T. C. Fogarty, and A. Munro. *Genetic Algorithms and Soft Computing*, volume 8 of *Studies in fuzziness and soft computing*, chapter Evolving Temporal Fuzzy Rule-Bases for Distributed Routing Control in Telecommunication Networks, pages 467–488. Physica-Verlag, 1996.

[54] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual Modelling of Workflows. In *OOER '95: Proceedings of the 14th International Conference on Object-Oriented and Entity-Relationship Modelling*, pages 341–354, London, UK, Dec. 1995. Springer-Verlag.

[55] J. Casillas, O. Cordón, , M. J. del Jesus, and F. Herrera. Genetic Tuning of Fuzzy Rule Deep Structures Preserving Interpretability and its Interaction with Fuzzy Rule Set Reduction. *IEEE Transactions on Fuzzy Systems*, 13(1):13–29, 2005.

[56] J. Casillas, O. Cordón, I. n. Fernández de Viana, and F. Herrera. Learning Cooperative Linguistic Fuzzy Rules Using the Best–Worst Ant System Algorithm: Research Articles. *Int. J. Intell. Syst.*, 20(4):433–452, 2005.

[57] J. Casillas, O. Cordón, and F. Herrera. COR: A Methodology to Improve Ad Hoc Data-Driven Linguistic Rule Learning Methods by Inducing Cooperation Among Rules. *IEEE Transactions on Systems, Man, and Cybernetics-Part B: Cybernetics*, 32(4):526–537, 2002.

[58] D. K. C. Chan, J. Vonk, G. Sánchez, P. W. P. J. Grefen, and P. M. G. Apers. A Specification Language for the WIDE Workflow Model. In *SAC '98: Proceedings of the 1998 ACM Symposium on Applied Computing*, pages 197–199, Atlanta, Georgia, United States, Feb. 1998. ACM Press.

[59] B. Chandrasekaran. Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert-System Design. *IEEE Expert*, 1(3):23–30, 1986.

[60] B. Chandrasekaran, T. R. Johnson, and J. W. Smith. Task-Structure Analysis for Knowledge Modeling. *Commun. ACM*, 35(9):124–137, 1992.

[61] P.-C. Chang, J.-C. Hsieh, and C.-Y. Wang. Adaptive Multi-Objective Genetic Algorithms for Scheduling of Drilling Operation in Printed Circuit Board Industry. *Applied Soft Computing*, 7(3):800–806, 2007.

[62] E. D. Chapple and L. R. Sayles. *The Measure of Management. Designing Organizations for Human Effectiveness.* New York : Macmillan, New York, USA, 1961.

[63] T. C. E. Cheng, Q. Ding, and B. M. T. Lin. A Concise Survey of Scheduling with Time-Dependent Processing Times. *European Journal of Operational Research*, 152:1–13, 2004.

[64] Y.-s. Cheng, Z.-j. Wang, C.-m. Wang, L.-y. Tang, and L. Shang. Modeling and Verifying Composite Semantic Web Service Based on Colored Petri Nets. In *Proceedings of the Sixth International Conference on Advanced Language Processing and Web Information Technology (ALPIT 2007)*, pages 510–514, Washington, DC, USA, Aug. 2007. IEEE Computer Society.

[65] Chin Soon Chong, Appa Iyer Sivakumar, Malcolm Yoke Hean Low, and Kheng Leng Gay. A Bee Colony Optimization Algorithm to job Shop Scheduling. In *WSC '06: Proceedings of the 38th Conference on Winter Simulation*, pages 1954–1961, Monterey, California, Dec. 2006. Winter Simulation Conference.

[66] I. Choi, C. Park, and C. Lee. A Transactional Workflow Model for Engineering/Manufacturing Processes. *Int. J. Computer Integrated Manufacturing*, 15(2):178–192, 2002.

[67] S. Chriestensen and N. D. Hansen. Coloured Petri Nets Extended with Channels for Synchronous Communication. In *15th Int. Conf. Application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*, pages 159–178, Zaragoza, Spain, June 1994. Springer-Verlag.

[68] S. Chriestensen and L. Petrucci. Modular Analysis of Petri Nets. *The Computer Journal*, 43(3):224–242, 2000.

[69] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *Web Services Description Language (WSDL).* World Wide Web Consortium (W3C), Mar. 2001. W3C recommendation.

[70] W. J. Clancey. The Epistemiology of a Rule-Based Expert System: A Framework for Explanation. *Artificial Intelligence*, 20(3):215–251, 1983.

[71] W. J. Clancey, P. Sachs, M. Sierhuis, and R. W. Hoof. BRAHMS: Simulating Practice for Work Systems Design. *International Journal of Human-Computer Studies*, 49(6):831–865, 1998.

[72] J. Clark, C. Casanave, K. Kanaskie, B. Harvey, J. Clark, N. Smith, J. Yunker, and K. Riemer. *ebXML Business Process Specification Schema 6 Version 1.01.* UN/CEFACT and OASIS, May 2001.

[73] J. Clark and M. Murata. RELAX NG Specification, 2001. OASIS Committee Specification.

[74] C. A. Coello. *Evolutionary Multiobjective Optimization: Theoretical Advances And Applications*, chapter Recent Trends in Evolutionary Multiobjective Optimization, pages 7–32. Springer-Verlag, 2005.

[75] O. Cordon and F. Herrera. A Two-Stage Evolutionary Process for Designing TSK Fuzzy Rule-Based Systems. *IEEE Transactions on Systems, Man and Cybernetics-Part B*, 29(6):703–715, 1999.

[76] O. Cordón and F. Herrera. Hybridizing Genetic Algorithms with Sharing Scheme and Evolution Strategies for Designing Approximate Fuzzy Rule-Based Systems. *Fuzzy sets and systems*, 118:235–255, 2001.

[77] O. Cordón, F. Herrera, F. Hoffmann, and L. Magdalena. *Genetic Fuzzy Systems: Evolutionary Tuning and Learning of Fuzzy Knowledge Bases*, volume 19 of *Advances in Fuzzy Systems - Applications and Theory.* World Scientific, 2001.

[78] D. Corne, J. D. Knowles, and M. J. Oates. The Pareto Envelope-Based Selection Algorithm for Multi-objective Optimisation. In *Proceedings of the 6th International Conference on Parallel Problem Solving from Nature*, pages 839–848, London, UK, Sept. 2000. Springer-Verlag.

[79] D. W. Corne, N. R. Jerram, J. D. Knowles, and M. J. Oates. PESA-II: Region-Based Selection in Evolutionary Multiobjective Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 283–290, San Francisco, California, USA, July 2001. Morgan Kaufmann.

[80] B. Curtis, M. I. Kellner, and J. Over. Process Modeling. *Commun. ACM*, 35(9):75–90, 1992.

[81] J. Dang, J. Huang, and M. N. Huhns. Workflow Coordination for Service-Oriented Multiagent Systems. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems (AAMAS '07)*, pages 1–3, New York, NY, USA, May 2007. ACM Press.

[82] H. Davulcu, M. Kifer, C. R. Ramakrishnan, and I. V. Ramakrishnan. Logic Based Modeling and Analysis of Workflows. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 25–33, Seattle, Washington, USA, June 1998. ACM.

[83] J. De Bruijn, C. Bussler, J. Domingue, D. Fensel, M. Hepp, U. Keller, M. Kifer, B. König Ries, J. Kopecky, R. Lara, H. Lausen, E. Oren, A. Polleres, D. Roman, J. Scicluna, and M. Stollberg. *Web Service Modeling Ontology (WSMO).* World Wide Web Consortium (W3C), June 2005. W3C member submission.

[84] M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel Schneider, and L. A. Stein. *OWL Web Ontology Language Reference*, Oct. 2004. W3C recommendation.

[85] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, Apr. 2002.

[86] W. Deiters. Information Gathering and Process Modeling in a Petri Net Based Approach. In *Business Process Management, Models, Techniques, and Empirical Studies*, pages 274–288, London, UK, Mar. 2000. Springer-Verlag.

[87] W. Deiters and V. Gruhn. The FUNSOFT Net Approach to Software Process Management. *International Journal of Software Engineering and Knowledge Engineering*, 4(2):229–256, 1994.

[88] C. Dellarocas and M. Klein. A Knowledge-Based Approach for Designing Robust Business Processes. In *Business Process Management, Models, Techniques, and Empirical Studies*, pages 50–65, London, UK, Mar. 2000. Springer-Verlag.

[89] Z. Ding, J. Wang, and C. Jiang. An Approach for Synthesis Petri Nets for Modeling and Verifying Composite Web Service. *J. Inf. Sci. Eng.*, 24(5):1309–1328, Dec. 2008.

[90] J. M. Dodero and Ernie Ghiglione. ReST-Based Web Access to Learning Design Services. *IEEE Transactions on Learning Technologies*, 1(3):190–195, 2008.

[91] M. Dumas and A. H. M. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. In *Proceedings of the International Conference on the Unified Modeling Language (UML)*, pages 76–90, Toronto, Canada, Oct. 2001. Springer Verlag.

[92] J. J. Durillo, A. J. Nebro, F. Luna, B. Dorronsoro, and E. Alba. jMetal: A Java Framework for Developing Multi-Objective Optimization Metaheuristics. Technical Report ITI-2006-10, Departamento de Lenguajes y Ciencias de la Computación, University of Málaga, E.T.S.I. Informática, Campus de Teatinos, Dec. 2006.

[93] S. Dustdar. Reconciling Knowledge Management and Workflow Management Systems: The Activity-Based Knowledge Management Approach. *Journal of Universal Computer Science*, 11(4):589–604, 2005.

[94] J. Eder and W. Liebhart. The Workflow Activity Model WAMO. In *OOER '95: Proceedings of the 14th International Conference on Object-Oriented and Entity-Relationship Modelling*, pages 87–98, Washington, DC, USA, Aug. 1998. IEEE Computer Society.

[95] H. Ehrig, G. Juhás, J. Padberg, and G. Rozenberg. *Unifying Petri Nets, Advances in Petri Nets*, volume 2128 of *Lecture Notes in Computer Science*. Springer-Verlag, London, UK, 2001.

[96] C. A. Ellis. Information Control Nets: A Mathematical Model of Office Information Flow. In *Proceedings of the Conference on Simulation, Measurement and Modeling of Computer Systems*, pages 225–240, Boulder, Colorado, Nov. 1979. ACM Press.

[97] C. A. Ellis and G. J. Nutt. Office information systems and computer science. *ACM Comput. Surv.*, 12(1):27–60, 1980.

[98] C. A. Ellis and G. J. Nutt. Office Information Systems and Computer Science. *ACM Comput. Surv.*, 12(1):27–60, 1980.

[99] J. P. Escobedo, L. de la Fuente Valentin, S. Gutierrez, A. Pardo, and C. Delgado Kloos. Implementation of a Learning Design Run-Time Environment for the .LRN Learning Management System. *Journal of Interactive Media in Education*, 1:1–12, 2007.

[100] R. Eshuis and R. Wieringa. A Formal Semantics for UML Activity Diagrams - Formalising Workflow Models. Technical Report CTIT-01-04, University of Twente, Department of Computer Science, 2001.

[101] H. Eskandari and C. D. Geiger. A Fast Pareto Genetic Algorithm Approach for Solving Expensive Multiobjective Optimization Problems. *Journal of Heuristics*, 14(3):203–241, 2008.

[102] R. Farahbod, U. Glässer, and M. Vajihollahi. Specification and Validation of the Business Process Execution Language for Web Services. In *Abstract State Machines 2004. Advances in Theory and Practice*, volume 3052 of *Lecture Notes in Computer Science*, pages 78–94. Springer Berlin / Heidelberg, 2004.

[103] D. Fensel and C. Bussler. The Web Service Modeling Framework WSMF. *Electronic Commerce: Research and Applications*, 1(2):113–137, 2002.

[104] D. Fensel, E. Motta, V. R. Benjamins, M. Crubezy, S. Decker, M. Gaspari, R. Groenboom, W. Grosso, M. Musen, E. Plaza, G. Schreiber, R. Studer, and B. Wielinga. The Unified Problem-solving Method Development Language UPML. *Knowledge and Information Systems*, 5(1):83–131, 2003.

[105] B. Fernández Gallego, M. Lama, J. C. Vidal, E. Sánchez, and A. Bugarín. Arquitectura Orientada a Servicios para la Ejecución de Unidades de Aprendizaje en Mundos Virtuales. In *Actas de las V Jornadas Científico-Técnicas en Servicios Web y SOA (JSWEB 2009)*, Madrid, Spain, Sept. 2009. Órbigo Printing Media.

[106] B. Fernández Gallego, M. Lama, J. C. Vidal, E. Sánchez, and A. Bugarín. OPENET VE: A Platform for the Execution of IMS LD Units of Learning in Virtual Environments. In *Proceedings of the 10th IEEE International Conferenceon Advanced Learning Technologies (ICALT 2010)*, Sousse, Tunisia, July 2010. IEEE Computer Society.

[107] P. Fodor. Initial Results on the F-logic to OWL Bi-directional Translation on a Tabled Prolog Engine. *CoRR*, abs/0808.1721:1–6, Aug. 2008.

[108] C. M. Fonseca and P. J. Fleming. Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization. In *Proceedings of the fifth International Conference on Genetic Algorithms (ICGA'93)*, pages 416–423, San Mateo, California, June 1993. IEEE Service Center.

[109] M. S. Fox and M. Gruninger. Enterprise Modelling. *AI Magazine*, 19(3):109–121, 1998.

[110] T. Freytag. WoPeD: Workflow Petri Net Designer. In *Proceedings of the 26th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ATPN'05)*, Miami, USA, June 2005.

[111] M. R. Garey, D. S. Johnson, and R. Sethi. The Complexity of Flowshop and Jobshop Scheduling. *Mathematics of Operational Research*, 1(2):117–129, 1976.

[112] D. Gasevic and V. Devedzic. Petri Net Ontology. *Knowledge-Based Systems*, 19(4):220–234, 2006.

[113] J. H. Gennari, M. A. Musen, R. W. Fergerson, W. E. Grosso, M. Crubézy, H. Eriksson, N. F. Noy, and S. W. Tu. The Evolution of Protégé: An Environment for Knowledge-Based Systems Development. *Int. J. Hum.-Comput. Stud.*, 58(1):89–123, 2003.

[114] D. Georgakopoulos, M. F. Hornick, and A. P. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[115] B. Giffler and G. L. Thompson. Algorithms for Solving Production Scheduling Problems. *Operations Research*, 8(4):487–503, 1960.

[116] Y. Gil, V. Ratnakar, E. Deelman, G. Mehta, and J. Kim. Wings for Pegasus: Creating Large-Scale Scientific Applications Using Semantic Representations of Computational Workflows. In *22th AAAI Conference on Artificial Intelligence*, pages 1767–1774, Vancouver, British Columbia, Canada, July 2007. AAAI Press.

[117] A. Giornada and F. Neri. Search-Intensive Concept Induction. *Evolutionary Computation*, 3(4):375–416, 1995.

[118] C. Girault and R. Valk. *Petri Nets for System Engineering*, chapter Workflow Systems, pages 507–539. Springer-Verlag, 2003.

[119] I. Global Learning Consortium. *IMS Learning Design Information Model*, Mar. 2003. Version 1.0 Final Specification.

[120] L. Gomes and J. P. Barro. Structuring and Composability Issues in Petri Nets Modeling. *IEEE Transactions on Industrial Informatics*, 1(2):112–123, 2005.

[121] A. Gómez Pérez, M. Fernández López, and O. Corcho. *Ontological Engineering*. Springer-Verlag, Nov. 2003.

[122] T. R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2):199–220, June 1993.

[123] S. K. Gupta and D. S. Nau. A Systematic Approach for Analyzing the Manufacturability of Machined Parts. *Computer Aided Design*, 27(5):323–342, 1995.

[124] Y. Gurevich. *Specification and Validation Methods*, chapter Evolving Algebras 1993: Lipari Guide, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.

[125] K. Hamada, T. Baba, K. Sato, and M. Yufu. Hybridizing a Genetic Algorithm with Rule-Based Reasoning for Production Planning. *IEEE Expert: Intelligent Systems and Their Applications*, 10(5):60–67, 1995.

[126] M. Hammer and J. Champy. *Reengineering the Corporation: A Manifesto for Business Revolution*. HarperBusiness, Apr. 1994.

[127] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

[128] E. Hart, P. Ross, and D. Corne. Evolutionary Scheduling: A Review. *Genetic Programming and Evolvable Machines*, 6(2):191–220, 2005.

[129] M. Havey. *Essential Business Process Modeling.* O'Reilly Media, Inc., 2005.

[130] M. Hepp, F. Leymann, J. Domingue, A. Wahler, and D. Fensel. Semantic Business Process Management: A Vision Towards Using Semantic Web Services for Business Process Management. *IEEE International Conference on E-Business Engineering*, 0:535–540, 2005.

[131] J. W. Herrmann and M. M. Chincholkar. Reducing Throughput Time during Product Design. *Journal of Manufacturing Systems*, 20(6):416–428, 2001.

[132] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In *Proceedings of the 3rd International Conference on Business Process Management (BPM 2005)*, pages 220–235, Nancy, France, Sept. 2005.

[133] F. Hoffmann and O. Nelles. Genetic Programming for Model Selection of TSK-Fuzzy Systems. *Information Sciences*, 136(1-4):7–28, 2001.

[134] D. Hollinsworth. The Workflow Reference Model. Technical Report TC00-1003, Workflow Management Coalition, 1994.

[135] A. W. Holt. Coordination Technology and Petri Nets. In *Advances in Petri Nets*, volume 222 of *Lecture Notes in Computer Science*, pages 278–296, Berlin, Nov. 1985. Springer-Verlag.

[136] J. Horn, N. Nafpliotis, and D. E. Goldberg. A Niched Pareto Genetic Algorithm for Multiobjective Optimization. In *Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence*, pages 82–87, Piscataway, New Jersey, June 1994. IEEE Service Center.

[137] I. Horrocks, P. Patel Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*, Nov. 2003. W3C member submission.

[138] M. Hsu and C. Kleissner. ObjectFlow: Towards a Process Management Infrastructure. *Distributed and Parallel Databases*, 4(2):169–194, 1996.

[139] S. Huang, Y. Hu, and C. Li. A TCPN Based Approach to Model the Coordination in Virtual Manufacturing organizations. *Computers and Industrial Engineering*, 47(1):61–76, 2004.

[140] P. Huber, K. Jensen, and R. M. Shapiro. Hierarchies in Colored Petri Nets. In *10th International Conference on Applications and Theory of Petri Nets*, pages 192–209, Bonn, Germany, June 1989.

[141] K. E. Huff. *Trends in Software Process*, chapter Software Process Modeling, pages 1–24. John Wiley & Sons, Apr. 1996.

[142] E. Hurink, B. Jurish, and M. Thole. Tabu Search for the Job Shop Scheduling Problem with Multi-Purpose machine. *Operations Research Spektrum*, 15:205–215, 1994.

[143] L. International and L. Foundation. *LAMS 2.3*. Macquarie University in Sydney Australia, Jan. 2010.

[144] ISO/IEC. Software and Systems Engineering - High-level Petri Nets Part 1: Concepts, Definitions and Graphical Notation, 2002. Final Draft of the International Standard ISO/IEC 15909-1.

[145] ISO/IEC. Software and Systems Engineering - High-level Petri Nets Part 2: Transfer Format, 2005. Working Draft of the International Standard ISO/IEC 15909-2.

[146] S. Jablonski. MOBILE: A Modular Workflow Model and Architecture. In *Proceedings of the Fourth International Working Conference on Dynamic Modelling and Information Systems*, pages 1–30, Noordwijkerhout, Netherlands, Sept. 1994.

[147] S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture and Implementation*. International Thomson Computer Press, Sept. 1996.

[148] A. S. Jain and S. Meeran. Deterministic Job-Shop Scheduling: Past, Present and Future. *European Journal of Operational Research*, 113:390–434, 1999.

[149] K. Jensen. *Coloured Petri Nets : Basic Concepts, Analysis Methods and Practical Use. Volume 1 (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, Apr. 2003.

[150] G. Kappel, P. Lang, S. Rausch Schott, and W. Retschitzegger. Workflow Management Based on Objects, Rules, and Roles. *IEEE Bulletin of the Technical Committee on Data Engineering*, 18(1):11–18, 1995.

[151] S. Kethers and M. Schoop. Reassessment of the Action Workflow Approach: Empirical Results. In *Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling (LAP 2000)*, pages 151–170, Aachen, Germany, Sept. 2000. Aachener Informatik Berichte.

[152] M. Kiefer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of ACM*, 42:741–843, 1995.

[153] B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD Dissertation, Queensland University of Technology, Brisbane, Australia, 2003.

[154] J. D. Knowles and D. W. Corne. Approximating the Nondominated Front Using the Pareto Archived Evolution Strategy. *Evolutionary Computation*, 8(2):149–172, 2000.

[155] A. Kö. Knowledge Modeling Techniques in Workflow Systems. In *14th International Workshop on Database and Expert Systems Applications (DEXA'03)*, page 823, Washington, DC, USA, Sept. 2003. IEEE Computer Society.

[156] E. Koper. *Modeling Units of Study from a Pedagogical Perspective: the Pedagogical Metamodel behind EML*, Oct. 2001.

[157] T. Kovacs. *Strength or Accuracy: Credit Assigment in Learning Classifier Systems*. Springer-Verlag, 2004.

[158] J. Koza. *Genetic Programming: on the Programming of Computers by means of Natural Selection*. The MIT Press, 1992.

[159] N. Krishnakumar and A. Sheth. Managing Heterogeneous Multi-System Tasks to Support Enterprise-wide Operations. *Distrib. Parallel Databases*, 3(2):155–186, 1995.

[160] S. Kukkonen and J. Lampinen. GDE3: The Third Evolution Step of Generalized Differential Evolution. In *Proceedings of the 2005 Congress on Evolutionary Computation (CEC 2005)*, pages 443–450, Edinburgh, Scotland, Sept. 2005. IEEE Service Center.

[161] A. Kusiak and W. He. Design of Components for Schedulability. *European Journal of Operational Research*, 164:185–194, 1999.

[162] C. A. Lakos and S. Christensen. A General Systematic Approach to Arc Extensions for Coloured Petri Nets. *Lecture Notes in Computer Science*, 815:338–357, 1994.

[163] M. Lama, J. C. Vidal, and A. Bugarín. Modelado de la Semántica Operacional de Procesos OWL-S a través de una Ontología de Redes de Petri. In *Actas de las II Jornadas Científico-Técnicas en Servicios Web (JSWEB 2006)*, pages 52–58. Unidixital S.L., Nov. 2006.

[164] Y. Lei and M. P. Singh. A Comparison of Workflow Metamodels. In *Proceedings of the ER-97 Workshop on Behavioral Modeling and Design Transformations: Issues and Opportunities in Conceptual Modeling*, Los Angeles, California, USA, Nov. 1997. Springer.

[165] D. Lei1 and Z. Wu1. Crowding-measure-Based Multiobjective Evolutionary Algorithm for Job Shop Scheduling. *International Journal of Advanced Manufacturing Technology*, 30(1-2):112–117, 2006.

[166] V. J. Leon, S. D. Wu, and R. H. Storer. Robustness Measures and Robust Scheduling for Job Shops. *IIE Transactions*, 26(5):32–43, 1994.

[167] K. S. Leung, Y. Leung, L. So, and K. F. Yam. Rule Learning in Expert Systems Using Genetic Algorithm: 1, Concepts. In *Proceedings of the 2nd International Conference on Fuzzy Logic and Neural Networks*, pages 201–204, Iizuka, Japan, June 1992.

[168] F. Leymann. *Web Services Flow Language (WSFL 1.0)*. IBM, May 2001.

[169] F. Leymann and W. Altenhuber. Managing Business Brocesses as an Information Resource. *IBM Syst. J.*, 33(2):326–348, 1994.

[170] H. Li and D. Zhun. Study of the reusable workflow system. *The Journal of American Science*, 1(2):51–60, 2005.

[171] C. J. Lin and Y. J. Xu. The Design of TSK-type Fuzzy Controllers Using a New Hybrid Learning Approach. *International Journal of Adaptive Control and Signal Processing*, 20(1):1, 2006.

[172] N. Liouane and I. Saad. Ant Systems & Local Search Optimization for Flexible Job Shop Scheduling Production. *International Journal of Computers, Communications & Control*, 2(2):174–184, 2007.

[173] D. E. Mahling, N. Craven, and W. B. Croft. From office automation to intelligent workflow systems. *IEEE Expert: Intelligent Systems and Their Applications*, 10(3):41–47, 1995.

[174] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. *OWL-S: Semantic Markup for Web Services*. World Wide Web Consortium (W3C), Nov. 2004. W3C member submission.

[175] M. Mastrolilli and L. M. Gambardella. Effective Neighborhood Functions for the Flexible Job Shop Problem. *Journal of Scheduling*, 3(1):3–20, 1998.

[176] R. Medina Mora, T. Winograd, R. Flores, and F. Flores. The Action Workflow Approach to Workflow Management Technology. In *Proceedings of the 1992 ACM Conference on Computer-Supported Cooperative Work (CSCW'92)*, pages 281–288, Toronto, Ontario, Canada, 1992. ACM.

[177] K. Mesghouni, S. Hammadi, and P. Borne. Evolutionary Algorithms for Job-Shop Scheduling. *International Journal of Applied Mathematics and Computer Science*, 14(1):93–103, 2004.

[178] H. Miao, T. He, and Z. Qian. Modeling and Analyzing Composite Semantic Web Service Using Petri Nets. In *Proceedings of the 2008 IEEE International Conference on e-Business Engineering (ICEBE '08)*, pages 660–664, Washington, DC, USA, Oct. 2008. IEEE Computer Society.

[179] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[180] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[181] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[182] I. Minis, J. W. Herrmann, G. Lam, and E. Lin. A Generative Approach for Concurrent Manufacturability Evaluation and Subcontractor Selection. *Journal of Manufacturing Systems*, 18(6):383–395, 1999.

[183] D. Moldt and J. Ortmann. DaGen: A Tool for Automatic Translation from DAML-S to High-Level Petri Nets. In *Fundamental Approaches to Software Engineering: 7th International Conference (FASE 2004)*, pages 209–213, Barcelona, Spain, Mar. 2004. Springer-Verlag.

[184] M. F. Möller. A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning. *Neural Netw.*, 6(4):525–533, 1993.

[185] B. Motik and R. Rosati. A Faithful Integration of Description Logics with Logic Programming. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 477–482, Hyderabad, India, Jan. 2007.

[186] E. Motta. An Overview of the OCML Modelling Language. In *8th Workshop on Knowledge Engineering Methods and Languages (KEML'98)*, pages 21–22, Karlsruhe, Germany, Jan. 1998.

[187] E. Motta. *Reusable Components for Knowledge Modelling: Case Studies in Parametric Design Problem Solving.* IOS Press, Amsterdam, The Netherlands, 1999.

[188] M. Mucientes, J. C. Vidal, A. Bugarín, and M. Lama. Processing Times Estimation in a Manufacturing Industry through Genetic Programming. In *Proceedings of the 3rd International Symposium on Genetic and Evolving Fuzzy Systems*, pages 95–100, Witten-Bommerholz, Germany, Mar. 2008. IEEE CS Press.

[189] M. Mucientes, J. C. Vidal, A. Bugarín, and M. Lama. Processing Times Estimation by Variable Structure TSK Rules Learned through genetic programming. *Soft Computing*, 13(5):497–509, Mar. 2009.

[190] T. Murata. Petri nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[191] S. Narayanan and S. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proceedings of the 11th International World Wide Web Conference (WWW'02)*, pages 77–88, Hawaii, USA, Nov. 2002. ACM Press.

[192] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. R. Swartout. Enabling Technology for Knowledge Sharing. *AI Magazine*, 12(3):36–56, 1991.

[193] A. Newell and H. A. Simon. Computer Science as Empirical Inquiry: Symbols and Search. *Commun. ACM*, 19(3):113–126, 1976.

[194] E. Nowicki and C. Smutnicki. A Fast Taboo Search Algorithm for the Job-Shop Scheduling Problem. *Management Science*, 42(6):797–813, 1996.

[195] P. O'Grady and R. E. Young. Issues in Concurrent Engineering Systems. *Journal of Design and Manufac-turing: The Research Journal of Concurrent Engineering*, 1:1–9, 1991.

[196] B. Omelayenko, M. CrubÃ©zy, D. Fensel, Y. Ding, E. Motta, and M. Musen. UPML Version 2.0. IBROW Deliverable D5, Free University of Amsterdam, 2000.

[197] OMG. Business Process Modeling Notation (BPMN) Specification. Technical Report formal/2008-01-17, Object Management Group, Jan. 2008. OMG Available Specification.

[198] W. M. P. Ouyang, C. van der Aalst, M. Dumas, and ter Hofstede. Translating BPMN to BPEL. Technical Report Technical Report, BPM-06-02, www.BPMcenter.org, 2006.

[199] M. Paolucci, A. Ankolekar, N. Srinivasan, and K. Sycara. The DAML-S Virtual Machine. In *2nd International Semantic Web Conference (ISWC 2003)*, pages 290–305, Sandial Island, FL, USA, Oct. 2003.

[200] S. E. Papadakis and J. B. Theocharis. A Genetic Method for Designing TSK Models Mased on Objective Weighting: Application to Classification Problems. *Soft Computing*, 10(9):805–824, 2006.

[201] G. Paquette and M. Léonard. The Educational Modeling of a Collaborative Game Using MOT+LD. In *Proceedings of the Sixth IEEE International Conference on Advanced Learning Technologies (ICALT 2006)*, Kerkrade, The Netherlands, July 2006. IEEE Computer Society.

[202] S. Petkov, E. Oren, and A. Haller. Aspects in Workflow Management. Technical Report DERI Technical Report 2005-04-10, Swiss Federal Institute of Technology (ETH), Apr. 2005.

[203] C. A. Petri. *Kommunikation mit Automaten.* PhD Dissertation, Institutes für Instrumentelle Mathematik, Germany, 1962.

[204] S. G. Ponnambalam, P. Aravindan, and P. Rao. Comparative Evaluation of Genetic Algorithms for Job-Shop Scheduling. *Production Planning and Control*, 12(6):560–574, 2001.

[205] S. G. Ponnambalam, V. Ramkumar, and N. Jawahar. A Multiobjective Genetic Algorithm for Job Shop Scheduling. *Production Planning and Control*, 12(8):764–774, 2001.

[206] A. R. Puerta, J. W. Egar, S. W. Tu, and M. A. Musen. A Multiple-Method Knowledge-Acquisition Shell for the Automatic Generation of Knowledge-Acquisition Tools. *Knowledge Acquisition*, 4(2):171–196, 1992.

[207] A. Rawlings, P. van Rosmalen, R. Koper, M. Rodríguez Artacho, and P. Lefrere. Survey of Educational Modelling Languages (EMLs), 2002. CEN Information Society Standardization System Learning Technologies Workshop.

[208] J. C. Recker and J. Mendling. On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages. In *Proceedings of the 18th International Conference on Advanced Information Systems Engineering (CAiSE 2006)*, pages 521–532, Luxembourg, Luxembourg, June 2006.

[209] H. A. Reijers. *Design and Control of Workflow Processes*. PhD thesis, Technische Universiteit Eindhoven, Eindhoven, Nederlands, 2002.

[210] H. A. Reijers. *Design and Control of Workflow Processes*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[211] W. Reisig. Petri Nets and Algebraic Specifications. *Theoretical Computer Science*, 80(1):1–34, 1991.

[212] D. Roman, J. de Bruijn, A. Mocan, H. Lausen, J. Domingue, C. Bussler, and D. Fensel. WWW: WSMO, WSML, and WSMX in a Nutshell. In *Proceedings of the First Asian Semantic Web Conference (ASWC 2006)*, pages 516–522, Beijing, China, Sept. 2006.

[213] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77–106, 2005.

[214] D. Roman, J. Scicluna, and J. Nitzsche. Ontology-Based Choreography. Technical Report WSMO Deliverable D14v0.1, DERI Innsbruck, 2007.

[215] N. Russel, W. M. P. van der Aalst, A. H. M. ter Hofstede, and P. Wohed. On the Suitability of UML 2.0 Activity Diagrams for Business Process Modelling. In *Proceedings of the Third Asia-Pacific Conference on Conceptual Modelling (APCCM2006)*, pages 95–104, Hobart, Australia, Oct. 2006. ACM Press.

[216] N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Workflow Data Patterns. Technical Report QUT Technical report, FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.

[217] N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Workflow Resource Patterns. Technical Report BETA Working Paper Series, WP 127, Eindhoven University of Technology, 2004.

[218] N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Workflow Data Patterns: Identification, Representation and Tool Support. In *Proceedings of the 24th International Conference on Conceptual Modeling (ER 2005)*, pages 353–368, Klagenfurt, Austria, Oct. 2005.

[219] N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Workflow Data Patterns: Identification, Representation and Tool Support. In *Proceedings of the 24th International Conference on Conceptual Modeling (ER)*, pages 353–368, Berlin, Oct. 2005.

[220] N. Russell, A. H. M. ter Hofstede, and W. M. P. van der Aalst. newYAWL: Specifying a Workflow Reference Language using Coloured Petri Nets. In *Proceedings of the Eighth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, Aarhus, Denmark, Aug. 2007.

[221] N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns : A Revised View. Technical Report BPM Center Report BPM-06-22, BPMcenter.org, 2006.

[222] N. Russell, W. M. P. van der Aalst, and A. H. M. ter Hofstede. Exception Handling Patterns in Process-Aware Information Systems. Technical Report BPM Center Report BPM-06-04, BPMcenter.org, 2006.

[223] N. Russell, W. M. P. van der Aalst, and A. H. M. ter Hofstede. Workflow Exception Patterns. In *Proceedings of the 18th International Conference on Advanced Information Systems Engineering (CAiSE 06)*, pages 288–302, Berlin, June 2006.

[224] N. Russell, W. M. P. van der Aalst, A. H. M. ter Hofstede, and D. Edmond. Workflow Resource Patterns: Identification, Representation and Tool Support. In *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, pages 216–232, Porto, Portugal, June 2005.

[225] N. Russell, W. M. P. van der Aalst, A. H. M. ter Hofstede, and D. Edmond. Workflow Resource Patterns: Identification, Representation and Tool Support. In *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, pages 216–232, Berlin, June 2005.

[226] W. Sadiq and M. E. Orlowska. On Capturing Process Requirements of Workflow-Based Business Information Systems. In *Proceedings of the 3rd International Conference on Business Information Systems (BIS99)*, pages 281–294, Poznan, Poland, Apr. 1999. Springer-Verlag.

[227] K. Salimifard and M. Wright. Petri Net-Based Modelling of Workow Systems: An Overview. *European Journal of Operational Research*, 134(3):664–676, 2001.

[228] G. Schreiber, H. Akkermans, A. Anjewierden, R. de Hoog, N. Shadbolt, W. V. de Velde, and B. Wierlinga. *Knowledge Engineering and Management: The CommonKADS Methodology.* MIT-Press, 1999.

[229] P. Senkul, M. Kifer, and I. H. Toroslu. A Logical Framework for Scheduling Workflows under Resource Allocation Constraints. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 694–705, Hong Kong, China, Aug. 2002. VLDB Endowment.

[230] D. Shabtay and G. Steiner. A Survey of Scheduling with Controllable Processing Times. *European Journal of Operational Research*, 155:1643–1666, 2007.

[231] K. Shah and N. Ripon. *Evolutionary Scheduling*, chapter An Evolutionary Approach for Solving the Multi-Objective Job-Shop Scheduling Problem, pages 165–195. Springer, Apr. 2007.

[232] R. M. Shapiro. XPDL 2.1 - Integrating Process Interchange & BPMN. Technical Report 2008, Workflow Management Coalition, Jan. 2008. WfMC White Papers.

[233] I. Singh, B. Stearns, and M. Johnson. *Designing Enterprise Application with the J2EE Platform*. Addison-Wesley, 1999.

[234] H. Smith and P. Fingar. Workflow Is Just a Pi Process. *BPTrends*, 1:1–36, Jan. 2004.

[235] E. Sánchez, M. Lama, R. R. Amorim, J. C. Vidal, and A. Novegil. On the Use of an IMS LD Ontology for Creating and Executing Units of Learning. *Journal of Interactive Media in Education*, 1:1–17, 2008.

[236] L. Songfeng, S. Chengfu, and M. Xinjian. Using E-Connection and Description Logic for Formalizing and Analyzing High-level Petri Net. In *Proceedings of the 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNACS 2007)*, pages 514–517, Timisoara, Romania, Sept. 2007.

[237] P. Sridhar and C. Rajendran. Scheduling in Flowshop and Cellular Manufacturing Systems with Multiple Objectives - A Genetic Algorithmic Approach. *Production Planning and Control*, 7(4):374–382, 1996.

[238] C. Stahl. A Petri Net Semantics for BPEL. Informatik-Berichte 188, Humboldt-Universität zu Berlin, July 2005.

[239] L. Steels. Ccomponents of Expertise. *AI Magazine*, 11(2):29–49, 1990.

[240] L. Steels. Components of Expertise. *AI Mag.*, 11(2):30–49, 1990.

[241] M. Stefik. *Introduction to Knowledge Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[242] V. Subramaniam, T. Ramesh, G. K. Lee, Y. S. Wong, and G. S. Hong. Job Shop Scheduling with Dynamic Fuzzy Selection of Dispatching Rules. *International Journal of Advanced Manufacturing Technology*, 16(10):759–764, 2000.

[243] M. Sugeno and G. Kang. Structure Identification of Fuzzy Model. *Fuzzy Sets and Systems*, 28:15–33, 1988.

[244] T. Takagi and M. Sugeno. Fuzzy Identification of Systems and its Application to Modeling and Control. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-15:116–132, 1985.

[245] H. Tamaki, E. Nishino, and Abe S. A Genetic Algorithm Approach to Multi-Objective Scheduling Problems with Earliness and Tardiness Penalties. In *Proceedings of the 1999 Congress on Evolutionary Computation*, pages 839–848, Washington D.C., USA, July 1999. IEEE Service Center.

[246] S. Thatte. *XLANG - Web Services for Business Process Design.* Microsoft, 2001.

[247] R. K. Thiagarajan, A. K. Srivastava, A. K. Pujari, and V. K. Bulusu. BPML: A Process Modeling Language for Dynamic Business Models. In *International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*, page 239, Los Alamitos, CA, USA, June 2002. IEEE Computer Society.

[248] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML-Schema Part 1: Structures Second Edition*, 2005. W3C recommendation.

[249] J. T. E. Timm and G. C. Gannod. Grounding and Execution of OWL-S Based Semantic Web Services. In *2008 IEEE International Conference on Services Computing (SCC '08)*, pages 588–592, Washington, DC, USA, July 2008. IEEE Computer Society.

[250] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[251] S. Uckum, S. Bagachi, and K. Kawamura. Managing Genetic Search in Job Shop Scheduling. *IEEE Expert: Intelligent Systems and Their Applications*, 8(5):15–24, 1993.

[252] M. Uschold and M. Gruninger. Ontologies: Principles, Methods, and Applications. *Knowledge Engineering Review*, 11(2):93–155, 1996.

[253] R. Vaessens. *Generalized Job Shop Scheduling: Complexity and Local Search.* PhD dissertation, Eindhoven University of Technology, Netherlands, 1995.

[254] W. M. P. van der Aalst. Three Good Reasons for Using a Petri-net-based Workflow Management System. In S. Navathe and T. Wakayama, editors, *Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96),*, pages 179–201, Cambridge, Massachusetts, Nov. 1996.

[255] W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[256] W. M. P. van der Aalst. Patterns and XPDL: A Critical Evaluation of the XML Process Definition Language. Technical Report QUT Technical report, FIT-TR-2003-06, Queensland University of Technology, Brisbane, 2003.

[257] W. M. P. van der Aalst and P. J. S. Berens. Beyond Workflow Management: Product-Driven Case Handling. In *International ACM SIGGROUP Conference on Supporting Group Work (GROUP 2001)*, pages 42–51, New York, NY, USA, Sept. 2001. ACM Press.

[258] W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, and P. Wohed. Pattern-Based Analysis of BPML (and WSCI). Technical Report QUT Technical Report, FIT-TR-2002-04, Queensland University of Technology, Brisbane, 2002.

[259] W. M. P. van der Aalst and A. H. M. Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.

[260] W. M. P. van der Aalst, A. H. M. Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(3):5–51, 2003.

[261] W. M. P. van der Aalst and A. H. M. ter Hofstede. Workflow Patterns: On the Expressive Power of (Petri-Net-Based) Workflow Languages. In *Proceedings of the Fourth International Workshop on Practical Use of Coloured Petri Nets and the CPN Tools*, Aarhus, Denmark, Aug. 2002.

[262] W. M. P. van der Aalst, A. H. M. ter Hofstede, and M. Weske. Business Process Management: A Survey. In *Proceedings of the International Conference on Business Process Management (BPM 2003)*, pages 1–12, Berlin, Germany, June 2003.

[263] W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data Knowl. Eng.*, 47(2):237–267, 2003.

[264] W. M. P. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems (Cooperative Information Systems)*. The MIT Press, Mar. 2004.

[265] P. J. M. van Laarhoven, E. H. L. Aarts, and J. K. Lenstra. Job Shop Scheduling by Simulated Annealing. *Operations Research*, 40(1):113–125, 1992.

[266] J. C. Vidal, M. Lama, and A. Bugarín. A High-level Petri Net Ontology Compatible with PNML. *Petri Net Newsletter*, 71:11–23, 2006.

[267] J. C. Vidal, M. Lama, and A. Bugarín. A Workflow Modeling Framework Enhanced with Problem-Solving Knowledge. In *Proceedings of the 10th International Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES 2006)*, volume 4253 of *LNAI*, pages 623–632, Bournemouth, UK, Sept. 2006. Springer.

[268] J. C. Vidal, M. Lama, and A. Bugarín. *Integrated Intelligent Systems for Engineering Design*, chapter Integrated Knowledge-Based System for Product Design in Furniture Estimate, pages 345–361. Frontiers in Artificial Intelligence and Applications. IOS Press, 2006.

[269] J. C. Vidal, M. Lama, and A. Bugarín. Petri net Semantics for OWL-S Service Choreography. In *Proceedings of the International Workshop on Formal Aspects of Business Processes and Web Services*, pages 7–20, Siedlce, Poland, June 2007.

[270] J. C. Vidal, M. Lama, and A. Bugarín. Service-Oriented Architecture for Knowledge-Enriched Workflows Modelling and Execution. In *Business Process and Services Computing (BPSC 2007)*, Lecture Notes in Informatics, pages 69–77, Leipzig, Germany, Sept. 2007.

[271] J. C. Vidal, M. Lama, and A. Bugarín. A Framework for Unifying Problem-Solving Knowledge and Workflow Modelling. In *AAAI 2008 Spring Symposium: AI Meets Business Rules and Process Management*, pages 105–110, San Francisco, EEUU, Mar. 2008. AAAI Press.

[272] J. C. Vidal, M. Lama, A. Bugarín, and S. Barro. Problem-Solving Analysis for the Budgeting Task in Furniture Industry. In *Proceedings of the 7th International Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES 2003)*, volume 4253 of *LNAI*, pages 1307–1313, Oxfork, UK, Sept. 2003. Springer.

[273] J. C. Vidal, M. Lama, A. Bugarín, and S. Barro. Workflow-Based Information System for Furniture Budgeting. In *Proceedings of the 2003 IEEE Conference on Emerging Technologies and Factory Automation*, pages 54–61, Lisbon, Portugal, Sept. 2003. IEEE Computer Society.

[274] J. C. Vidal, M. Lama, A. Bugarín, R. Lago, and A. Novegil. Arquitectura Orientada a Servicios para el Modelado y la Ejecución de Flujos de Trabajo Enriquecidos con Conocimiento. In *Actas de las III Jornadas Científico-Técnicas en Servicios Web y SOA (JSWEB 2007)*, pages 19–26, Zaragoza, Spain, Sept. 2007. International Thomson Editores.

[275] J. C. Vidal, M. Lama, A. Bugarín, and M. Mucientes. Application of a Knowledge-Based Workflow Framework for Modelling the Price Estimation Task in Furniture Industry. In *Proceedings of the Twenty Third International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA-AIE 2010)*, Córdoba, Spain, June 2010.

[276] J. C. Vidal, M. Lama, and A. Bugarín. OPENET: Ontology-Based Engine for High-Level Petri Nets. *Expert Systems with Applications*, 37(9):6493–6509, Sept. 2010.

[277] J. C. Vidal, M. Lama, and A. Bugarín. Toward the Use of Petri Nets for the Formalization of OWL-S Choreographies. *Knowledge and Information Systems*, Article under Review, 2010.

[278] J. C. Vidal, M. Lama, A. Novegil, and A. Bugarín. Diseño e Implementación de un Motor de Ejecución de Coreografías de Servicios Web Semánticos Basado en Ontologías. In *Actas de las IV Jornadas Científico-Técnicas en Servicios Web y SOA (JSWEB 2008)*, pages 19–26, Sevilla, Spain, Oct. 2008.

[279] J. C. Vidal, M. Lama, E. Sánchez, and A. Bugarín. Application of Petri Nets on the Execution of IMS Learning Design Documents. In *Proceedings of the 3rd European Conference on Technology Enhanced Learning (EC-TEL 08)*, pages 95–100, Maastricht, Holand, Sept. 2008. Springer.

[280] J. C. Vidal, M. Lama, E. Sánchez, A. Bugarín, and A. Novegil. OPENET LD: A Petri Net-Based Engine to Execute Units of Learning Represented in IMS LD. In *Proceedings of the 9th IEEE International Conferenceon Advanced Learning Technologies (ICALT 2009)*, pages 499–503, Riga, Latvia, July 2009. IEEE Computer Society.

[281] J. C. Vidal, M. Mucientes, A. Bugarín, and M. Lama. Machine Scheduling in Custom Furniture Industry through Neuro-Evolutionary Hybridization. *Applied Soft Computing*, Article in Press, Corrected Proof doi:10.1016/j.asoc.2010.04.020, May 2010.

[282] J. C. Vidal, M. Mucientes, A. Bugarín, M. Lama, and R. S. Balay. Hybrid Approach for Machine Scheduling Optimization in Custom Furniture Industry. In *8th International Conference on Hybrid Intelligent Systems*, pages 849–854, Barcelona, Spain, Sept. 2008. IEEE Computer Society Press. This paper was awarded with the *Industry Best Paper Application Award* at the 8th International Conference on Hybrid Intelligent Systems (HIS2008).

[283] J. C. Vidal, M. Mucientes, M. Lama, and A. Bugarín. A Fuzzy Evolutionary Algorithm for Scheduling in Wood-Based Furniture Manufacturing. In Grupo de Investigación Oreto, editor, *XIII Congreso Español sobre Tecnologías y Lógica Fuzzy*, pages 217–222, Ciudad Real, Spain, Sept. 2006. Artes Gráficas Lince.

[284] J. C. Vidal, M. Mucientes, M. Lama, and A. Bugarín. An Adaptive Evolutionary Algorithm for Production Planning in Wood Furniture Industry. In P. Angelov, D. Filev, N. Kasabov, and O. Cordón, editors, *2006 International Symposium on Evolving Fuzzy Systems*, pages 267–273, Ambleside, UK, Sept. 2006. IEEE Computer Society. This paper was awarded with the *Runner-up Best Industry-Orientated Paper* at the 2006 International Symposium on Evolving Fuzzy Systems.

[285] J. C. Vidal, A. Novegil, M. Lama, and E. Sánchez. Service Oriented Architecture to Manage Units of Learning Based on the IMS LD Specification. In *EUNIS 2008: VISION IT*, pages 115–115, Ärhus, Denmark, June 2008. University of Ärhus.

[286] H. Vogten and H. Martens. *CopperCore Version 3.3*. Open University of Netherlands, Dec. 2009.

[287] C. Wallace. *Specification and Validation Methods*, chapter The Semantics of the C++ Programming Language, pages 131–164. Oxford University Press, Inc., New York, NY, USA, 1995.

[288] J. Wang. *Timed Petri Nets: Theory and Application*. Kluwer Academic Publishers, Boston, Massachusetts, USA, 1998.

[289] L.-X. Wang and J. M. Mendel. Generating Fuzzy Rules by Learning from Examples. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(6):1414–1427, 1992.

[290] Y. Wang, X. Bai, J. Li, and R. Huang. Ontology-Based Test Case Generation for Testing Web Services. In *Proceedings of the 8th International Symposium on Autonomous Decentralized Systems (ISADS 2007)*, pages 43–50, Sedona, Arizona, USA, Mar. 2007.

[291] J.-P. Watson, J. C. Beck, A. E. Howe, and L. D. Whitley. Problem Difficulty for Tabu Search in Job-Shop Scheduling. *Artificial Intelligence*, 143(2):189–217, 2003.

[292] M. Weber and E. Kindler. *Petri Net Technology for Communication-Based Systems*, volume 2472 of *Lecture Notes in Computer Science*, chapter The Petri Net Markup Language, pages 124–144. Springer-Verlag, 2003.

[293] WfMC. The Workflow Reference Model. Technical Report WFMC-TC-1003, Workflow Management Coalition, Jan. 1995. Document Status Issue 1.1.

[294] WfMC. Workflow Management Coalition Terminology Glossary. Technical Report WFMC-TC-1011, Workflow Management Coalition, Feb. 1999. Document Status Issue 3.0.

[295] WfMC. Process Definition Interface – XML Process Definition Language, version 1.15. Technical Report WFMC-TC-1025, Workflow Management Coalition, Oct. 2005. Document Status Final.

[296] B. J. Wielinga, A. T. Schreiber, and J. A. Breuker. KADS: a Modelling Approach to Knowledge Engineering. *Knowl. Acquis.*, 4(1):5–53, 1992.

[297] T. Winograd and F. Flores. *Understanding Computers and Cognition: A New Foundation for Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[298] D. Wodtke, J. Weissenfels, G. Weikum, and A. K. Dittrich. The Mentor Project: Steps Toward Enterprise-Wide Workflow Management. In *Proceedings of the Twelfth International Conference on Data Engineering (ICDE'96)*, pages 556–565, Washington, DC, USA, 1996. IEEE Computer Society.

[299] P. Wohed, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. Pattern-Based Analysis of BPEL4WS. Technical Report QUT Technical report, FIT-TR-2002-05, Queensland University of Technology, Brisbane, 2002.

[300] P. Wohed, W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, and N. Russel. Pattern-Based Analysis of the Control-Flow Perspective of UML Activity Diagrams. In *Proceedings of the 24th International Conference on Conceptual Modeling (ER 2005)*, pages 63–78, Berlin, Germany, Oct. 2005. Springer Verlag.

[301] P. Wohed, W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, and N. Russel. On the Suitability of BPMN for Business Process Modelling. In *Proceedings of the 4th International Conference on Business Process Management (BPM 2006)*, pages 161–176, Berlin, Germany, Sept. 2006. Springer Verlag.

[302] P. Wohed, W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, and N. Russel. Pattern-Based Analysis of BPMN - An Extensive Evaluation of the Control-flow, the Data and the Resource Perspectives (revised version). Technical Report BPM Center Report BPM-06-17, BPMcenter.org, 2006.

[303] M. L. Wong, W. Lam, K. S. Leung, P. S. Ngan, and J. C. Y. Cheng. Discovering Knowledge from Medical Databases using Evolutionary Algorithms. *IEEE Engineering in Medicine and Biology Magazine*, 19(4):45–55, 2000.

[304] L.-N. Xing, Y.-W. Chen, and K.-W. Yang. Multi-Objective Flexible Job Shop Schedule: Design and Evaluation by Simulation Modelling. *Applied Soft Computing*, 9(1):362–376, 2009.

[305] Q. Xu, R. Qiu, and F. Xu. Integration of Workflow and Multi-agents for Supply Chain Coordination. *Computer Engineering*, 29(15):19–21, 2003.

[306] G. Xue, J. Lu, and S. Yao. Investigating Workflow Patterns in Term of Pi-calculus. In *11th International Conference on Computer Supported Cooperative Work in Design, CSCWD 2007.*, pages 823–827, Melbourne, Vic., Apr. 2007.

[307] G. Yang, M. Kifer, and C. Zhao. FLORA-2: A Rule-Based Knowledge Representation and Inference Infrastructure for the Semantic Web. In *Proceedings of the Second International Conference on Ontologies, Databases and Applications of Semantics (ODBASE 2003)*, pages 671–688, Catania, Sicily, Italy, Nov. 2003.

[308] S. J. H. Yang, B. C. W. Lan, and J.-Y. Chung. A New Approach for Context Aware SOA. In *Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05)*, pages 438–443, Washington, DC, USA, Mar. 2005. IEEE Computer Society.

[309] J. Yen, L. Wang, and C. W. Gillespie. Improving the Interpretability of TSK Fuzzy Models by Combining Global Learning and Local Learning. *IEEE Transactions on Fuzzy Systems*, 6(4):530–537, 1998.

[310] M. Zisman. *Representation, Specification and Automation of Office Procedures.* PhD thesis, University of Pennsylvania, Warton School of Business, Pennsylvania, USA, 1977.

[311] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical Report 103, Swiss Federal Institute of Technology (ETH), 2001.

[312] E. Zitzler and L. Thiele. Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.

[313] M. Zur. Organizational Management in Workflow Applications - Issues and Perspectives. *Information Technology and Management*, 5(3):271–291, 2004.

[314] M. zur Muehlen and M. Gille. *Workflow Handbook 2002*, chapter Workflow Application Architectures - Classification and Characteristics of Workflow-Based Information Systems, pages 39–50. Workflow Management Coalition, Lighthouse Point, FL, USA, 2001.

[315] M. Zweben and M. S. Fox. *Intelligent Scheduling*. Morgan Kaufmann, Mar. 1994.