UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

Centro Singular de Investigación en Tecnoloxías da Información (CiTIUS)

Tesis doctoral

A GRAPH-BASED FRAMEWORK FOR OPTIMAL SEMANTIC WEB SERVICE COMPOSITION

Presentada por:

Pablo Rodríguez Mier

Dirigida por:

Manuel Lama Penín
Manuel Mucientes Molina

Santiago de Compostela, marzo de 2016

Manuel Lama Penín, Profesor Titular de Universidad del Área de Ciencia de la Computación e Inteligencia Artificial de la Universidade de Santiago de Compostela

Manuel Mucientes Molina, 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 **A GRAPH-BASED FRAMEWORK FOR OPTIMAL SEMANTIC WEB SERVICE COMPOSITION** ha sido realizada por **Pablo Rodríguez Mier** bajo nuestra dirección en el Centro Singular de Investigación en Tecnoloxías da Información de la Universidade de Santiago de Compostela, y constituye la Tesis que presenta para obtar al título de Doctor.

Santiago de Compostela, marzo de 2016

Manuel Lama Penín
Director de la tesis

Manuel Mucientes Molina
Director de la tesis

Pablo Rodríguez Mier
Autor de la tesis

Manuel Lama Penín, Profesor Titular de Universidad del Área de Ciencia de la Computación e Inteligencia Artificial de la Universidade de Santiago de Compostela

Manuel Mucientes Molina, Profesor Contratado Doctor del Área de Ciencia de la Computación e Inteligencia Artificial de la Universidade de Santiago de Compostela

como directores de la tesis titulada:

A GRAPH-BASED FRAMEWORK FOR OPTIMAL SEMANTIC WEB SERVICE COMPOSITION

Por la presente DECLARAN:

Que la tesis presentada por Don **Pablo Rodríguez Mier** es idónea para ser presentada, de acuerdo con el artículo 41 del *Regulamento de Estudos de Doutoramento*, por la modalidad de compendio de ARTÍCULOS, en los que el doctorando ha tenido participación en el peso de la investigación y su contribución fue decisiva para llevar a cabo este trabajo. Y que está en conocimiento de los coautores, tanto doctores como no doctores, participantes en los artículos, que ninguno de los trabajos reunidos en esta tesis serán presentados por ninguno de ellos en otras tesis de Doctorado, lo que firmamos bajo nuestra responsabilidad.

Santiago de Compostela, marzo de 2016

Manuel Lama Penín
Director de la tesis

Manuel Mucientes Molina
Director de la tesis

If an experiment works, something has gone wrong.

Finagle's First Law

Agradecimientos

En primer lugar, me gustaría agradecer a mis directores de tesis Manuel Lama y Manuel Mucientes por la confianza depositada en mí y por la ayuda ofrecida a lo largo de estos años, sin la cual esta tesis no hubiera sido posible. Gracias a ellos he tenido la oportunidad de introducirme en el mundo de la investigación, algo que desde hacía mucho tiempo siempre había querido explorar.

También me gustaría agradecer al Departamento de Electrónica y Computación y al Centro Singular de Investigación en Tecnoloxías da Información (CiTIUS), de la Universidade de Santiago de Compostela, por el entorno ofrecido, donde he tenido el placer de desarrollar gran parte de esta tesis, y en especial al director comisario del CiTIUS, Paulo Félix, por tener siempre un momento disponible para escucharnos y aconsejarnos, y también por apoyar muchas de nuestras iniciativas por muy excéntricas que fueran a veces algunas de ellas.

Quiero expresar también mi gratitud a las entidades que han financiado esta tesis. Concretamente, al Ministerio de Educación, Cultura y Deporte, a través del programa de Formación de Profesorado Universitario (FPU AP2010-1078), y a la fundación Pedro Barrié de la Maza, por su apoyo a mi investigación mediante la concesión de una beca predoctoral que me ha permitido realizar una estancia de investigación en el *Knowledge Media Institute* (KMi), en la universidad inglesa *The Open University*.

Precisamente, aprovecho para agradecer a la gente del KMi la cálida acogida que me han brindado, con los cuales he compartido muy buenos momentos. Gracias especialmente al Dr. Carlos Pedrinaci, con el que he tenido el placer de trabajar, por la fructífera colaboración y por la supervisión de mi trabajo durante la estancia (¡y por perfeccionar el *parabolic shoot*!).

Una mención especial va dedicada a todos los compañeros del CiTIUS que han hecho que mi etapa como doctorando haya merecido la pena. Particularmente, gracias a Adrián, Ana, Ángel, Arturo, Borja, David, Ismael, Jano, Jorge, Tefa, Tino y Tomás, que me han acom-

pañado a lo largo de este recorrido dentro y también fuera del CiTIUS, con los que me lo he pasado muy bien y de los cuales también he aprendido mucho. Han sido muchos años juntos compartiendo fracaso tras fracaso (¿en eso consistía la investigación no?) pero también alguna que otra pequeña victoria. También va dedicada a toda la gente de Vigo que ha estado todo este tiempo ahí disponible para pasar muy buenos ratos.

Gracias de paso a *StackOverflow* y a *Google Scholar* por contribuir a acortar la duración de la tesis.

Finalmente, y no por ello menos importante, agradecer a mi familia por su apoyo incondicional y por todo el cariño recibido. Concretamente a mis abuelos, por preocuparse siempre por mí y por interesarse en mi trabajo, aunque nunca supe explicarles muy bien en qué consistía. A mis padres, a los que les debo todo, quienes junto a mi hermana han estado siempre a mi lado, apoyándome y ayudándome a alcanzar mis metas. También a la familia de los Otero y los García por portarse tan bien conmigo. Y finalmente, a Patri, por todas las experiencias compartidas durante este largo camino, que nunca serán olvidadas.

Santiago de Compostela, marzo de 2016

Resumen

La Computación Orientada a Servicios (*Service Oriented Computing* o SOC en inglés) se ha convertido en el paradigma principal para el desarrollo de aplicaciones distribuidas. A diferencia del paradigma monolítico tradicional, donde las aplicaciones se conciben como procesos centrales gestionados por organizaciones individuales, el paradigma SOC se apoya en el concepto de los servicios como los componentes básicos de construcción para el desarrollo de aplicaciones distribuidas, facilitando tanto la construcción, como el mantenimiento o la reutilización mediante composición.

Los servicios son componentes software débilmente acoplados, que exponen una funcionalidad claramente definida a través de interfaces estándar y con capacidad de interoperabilidad a través de la red. Dado que los servicios son componentes independientes y autocontenidos, tienen la ventaja de que pueden ser fácilmente publicados, localizados e invocados a través de la red, lo que permite el desarrollo de nuevas aplicaciones mediante la explotación de la colaboración y la integración heterogénea de servicios entre distintas organizaciones.

En la práctica, este modelo conceptual para el desarrollo de aplicaciones distribuidas usando los servicios como principales actores se implementa mediante lo que se conoce como Arquitecturas Orientadas a Servicios (*Service Oriented Architectures* o SOA en inglés). Una arquitectura SOA es un conjunto de principios, patrones y criterios de diseño orientados a la construcción de sistemas distribuidos mediante el uso de servicios alineados con los procesos de negocio con el objetivo de mejorar la escalabilidad y flexibilidad de las organizaciones.

En general, en una SOA se identifican tres tipos de roles o participantes. Dichos roles son: 1) el *proveedor de servicios*, responsable de proporcionar la implementación de los servicios así como de la definición y publicación de su descripción; 2) el *cliente*, que puede ser tanto un usuario como otro componente software y desempeña el rol de consumidor de servicios, a través del descubrimiento del servicio adecuado y su posterior ejecución, con el objetivo de

satisfacer una necesidad concreta; y 3) el *registro de servicios*, cuya función es la de mantener localizada la información de los servicios publicados por los proveedores, además de facilitar los mecanismos necesarios para el descubrimiento de dichos servicios por parte de los clientes.

De la misma forma que los servicios son la piedra angular del paradigma SOC, los *servicios Web* son la tecnología más usada a la hora de implementar servicios en una SOA. Los servicios Web son aplicaciones modulares descritas por una serie de operaciones accesibles a través de la red soportadas por un conjunto de capas (pila de especificaciones) que definen los servicios a distintos niveles de abstracción, como son: la *capa de transporte*, encargada de la comunicación a nivel de red; la *capa de mensajería*, responsable de los protocolos de comunicación necesarios para la codificación y el intercambio de mensajes entre las distintas partes (XML-RPC, SOAP...); la capa de *calidad de servicio*, encargada de proporcionar información relevante sobre las características no funcionales de los servicios como pueden ser el tiempo de respuesta, la disponibilidad, el coste...; la *capa de descripción*, que especifica la forma en la que se describen las características funcionales de los servicios mediante el uso del lenguaje WSDL (Web Service Description Language), etc.

A nivel funcional, la capa de descripción es interesante ya que define los elementos necesarios para poder invocar un servicio, es decir, qué entradas y tipos de datos requiere la función implementada por el servicio, y qué datos devuelve tras la ejecución. Sin embargo, una limitación importante de esta capa es que sólo describe la funcionalidad desde un punto de vista sintáctico, es decir, se define la estructura y los tipos de las funciones que implementa un servicio pero no el significado de éstos. Esta limitación dificulta la automatización de tareas críticas tales como el descubrimiento, la composición o la invocación de servicios. Para superar esta limitación, se han llevado a cabo diversos esfuerzos en aras de proporcionar información semántica a las descripciones de los servicios. Como resultado, diversos lenguajes de anotación semántica para servicios como SAWSDL, WSMO u OWL-S han sido creados, dando paso a un nuevo tipo de servicios denominados servicios Web semánticos.

Los servicios Web semánticos se consideran una extensión de los servicios tradicionales que incorporan anotaciones semánticas con el fin de proporcionar definiciones declarativas formales de sus interfaces, así como para capturar de forma declarativa la funcionalidad de los servicios. Sobre la base de estas descripciones semánticas se establecen nuevos mecanismos para automatizar las tareas implicadas en el ciclo de vida de aplicaciones orientadas a servicios, por ejemplo, permitiendo el razonamiento acerca de los tipos de datos de las entradas

y salidas para mejorar el descubrimiento de servicios o para automatizar el encadenamiento de múltiples servicios mediante el correcto emparejamiento de sus entradas y salidas con el fin de generar nuevas funcionalidades mediante la creación de composiciones de servicios complejas.

Precisamente, una de las promesas clave de la computación orientada a servicios es la capacidad para generar nuevas aplicaciones distribuidas a bajo coste mediante la reutilización y combinación de servicios existentes. A este proceso de creación de nuevos servicios se le denomina *composición de servicios*. La composición de servicios conduce a la creación de nuevos servicios compuestos bajo demanda mediante la combinación de las entradas y salidas de los servicios existentes de una manera que este nuevo servicio cumpla alguna funcionalidad específica que no puede ser satisfecha por un único servicio aislado.

Aunque las composiciones pueden ser diseñadas manualmente, esta aproximación tiene limitaciones importantes. Por una parte, el diseño manual de una composición de servicios es una tarea compleja que requiere mucho esfuerzo para localizar los servicios adecuados y descubrir la mejor forma de combinarlos para conseguir obtener la funcionalidad buscada de manera óptima. A pesar de que el diseño manual es factible en dominios concretos, donde el número de servicios es limitado y relativamente estable, no es práctico cuando se trabaja con grandes volúmenes de servicios. Por otra parte, este enfoque requiere de la anticipación por parte de los implementadores de servicios a las posibles necesidades de los clientes y casos de uso comunes con el fin de diseñar las composiciones apropiadas que satisfagan dichas demandas. Esto limita en gran parte la flexibilidad de las organizaciones, dado que es prácticamente imposible prever las necesidades de todos los usuarios con antelación, y menos aún diseñar, depurar y desplegar múltiples composiciones para cada posible demanda. Además, el fallo en algún servicio de una composición puede provocar un incorrecto funcionamiento o un fallo total en la composición, lo que requiere más mano de obra para localizar el servicio afectado y buscar la forma de reemplazarlo por uno o por varios servicios que suplan dicha funcionalidad. Esto se traduce en paradas en los servicios que pueden impactar de forma negativa el normal funcionamiento de las organizaciones.

Estas limitaciones pueden ser superadas mediante el uso de técnicas automáticas para la composición de servicios Web. Sin embargo, la automatización de todo el proceso de composición es una tarea compleja que requiere no sólo de la automatización de muchas actividades necesarias para la composición, como el emparejamiento automático de entradas y salidas o el descubrimiento de servicios relevantes para la composición, sino también de la optimiza-

ción de los diferentes aspectos que afectan a la calidad de las soluciones, como pueden ser el número total de servicios o la calidad de servicio de dicha composición (por ejemplo, el tiempo de respuesta total o el número de peticiones por segundo que soporta).

Los beneficios de la composición automática de servicios han motivado la aparición de muchas propuestas distintas centradas en diferentes aspectos de la problemática y bajo diversas asunciones. En general, las técnicas existentes pueden clasificarse en *métodos basados en plantilla* si el *workflow* de composición viene determinado por una plantilla con un número fijo de servicios que el usuario debe proporcionar al sistema, o *basados en búsqueda* si toda la composición es creada de forma automática y por tanto el número de servicios puede ser variable. Un *workflow*, en el contexto de la composición de servicios, define un conjunto de tareas abstractas junto con las dependencias de control y de datos entre ellas, donde cada tarea especifica una funcionalidad abstracta que puede ser implementada por algún servicio concreto de entre todos los posibles candidatos para dicha tarea.

En los métodos basados en plantilla, el *workflow* de la composición es definido previamente por un experto, responsable de identificar las distintas tareas abstractas y modelar las dependencias de datos y control entre ellas. Una vez el *workflow* está diseñado, el sistema de composición automático se encarga de seleccionar el servicio más adecuado para cada tarea intentando optimizar uno o varios parámetros de la composición, generalmente referidos a propiedades no funcionales de calidad de servicio, como el coste, el tiempo de respuesta total, etc. Por tanto, los problemas que se abordan en este tipo de composiciones son principalmente: 1) cómo automatizar y realizar un descubrimiento adecuado de los servicios de forma eficiente y/o 2) cómo hacer la selección de servicios para cada tarea de forma que se optimice la calidad de servicio global sujeto a una o varias restricciones impuestas por el cliente sobre los parámetros deseados.

Aunque estas aproximaciones son capaces de buscar composiciones óptimas soportando restricciones complejas para múltiples parámetros, también tienen importantes limitaciones. Por una parte, dado que el *workflow* de composición está predefinido, el número de servicios está ligado al número de tareas abstractas definidas en el *workflow*, lo que impide optimizar el propio tamaño de la composición. Por otra parte, si una tarea no tiene ningún servicio válido que pueda implementarla, no es posible generar una composición alternativa que pueda reemplazar dicha tarea sin afectar al funcionamiento esperado. Por contra, en los métodos basados en búsqueda, todo el *workflow* de composición es generado de manera automática, incluyendo tanto las dependencias de control como de datos necesarias para la correcta ejecución de la

composición, lo que permite generar composiciones de tamaño y estructura variable. Este tipo de técnicas son más potentes pero computacionalmente más costosas, dado que, en general, el número de posibles composiciones y de formas de combinar los servicios crece de manera exponencial con el número de servicios disponibles. Este tipo de técnicas se pueden clasificar en al menos dos grandes bloques: centradas en control (*control-centric*) o centradas en datos (*data-centric*).

El objetivo de las aproximaciones centradas en control es el de buscar la mejor forma de combinar la funcionalidad de los servicios adecuados con las estructuras de control (como pueden ser ejecución en secuencia, ejecución en paralelo, bucles, etc) usadas para coordinar el flujo de control de la composición resultante, mientras que el enfoque centrado en los datos analiza las dependencias de datos entre las entradas y salidas de los servicios (es decir, qué salidas pueden ser potencialmente usadas como entradas de otros servicios) y luego infieren las dependencias de control a partir de dichas dependencias de datos. En general, los enfoques centrados en control son computacionalmente más difíciles ya que el uso de diferentes construcciones de control incrementa la variabilidad de las soluciones, así como el número de composiciones funcionalmente equivalentes para el mismo objetivo. Este aumento de la complejidad juega en contra de la capacidad de obtener soluciones razonables en poco tiempo. Por contra, los enfoques centrados en datos son más fáciles de generar y validar, pero a costa de una menor expresividad ya que no todas las posibles construcciones de control pueden inferirse simplemente mediante el análisis de dependencias de datos. Cada estrategia tiene sus propias ventajas y limitaciones, y su eficacia dependerá de las necesidades particulares y el contexto específico en el que la composición se llevará a cabo.

La mayor parte de las técnicas centradas en control para la generación automática de composiciones se basan en algoritmos evolutivos y en redes jerárquicas de tareas para planificación (*Hierarchical Task Networks* o HTNs en inglés). Las técnicas evolutivas, y concretamente la programación genética, son especialmente adecuadas ya que ofrecen un buen rendimiento en la optimización de problemas complejos con espacios de búsqueda de gran tamaño. La programación genética es un tipo de algoritmo evolutivo que genera y evoluciona de forma estocástica una población de individuos (soluciones) por medio de mecanismos de selección natural. La información de cada individuo se codifica a través del genotipo: una colección de genes que codifican las características de los individuos. La principal diferencia con respecto a otro tipo de algoritmos evolutivos es que la programación genética utiliza estructuras tipo árbol para representar el genotipo de los individuos, así como operadores genéticos que

operan sobre dichos árboles para transformar el genotipo de los individuos en el proceso de optimización. Estas estructuras en forma de árbol utilizadas para codificar las instancias del problema guardan mucha semejanza con la forma en que se definen los *workflows* de composición mediante la combinación recursiva de estructuras de control y servicios.

Del mismo modo, las HTNs utilizan una descomposición jerárquica de tareas de arriba abajo para generar automáticamente composiciones pero de una manera completamente diferente. En lugar de evolucionar una serie de composiciones generadas aleatoriamente, las HTNs aplican ciertos operadores de dominio específico que permiten descomponer los objetivos de composición a alto nivel planteados por un usuario (como podría ser la planificación de un viaje) en tareas de más bajo nivel (como podría ser reserva de un vuelo) hasta que dichas tareas puedan ser implementadas directamente por servicios Web. Las HTNs comparten algunas similitudes con los métodos basados en plantillas ya que se les debe proporcionar una descripción de las diferentes tareas y métodos de descomposición de dichas tareas, las cuales deben ser modeladas con antelación por un experto. Pero independientemente de la estrategia adoptada, existe una falta clara de estudios que analicen cómo se pueden generar composiciones óptimas, es decir, cómo generar composiciones de servicios de forma que se optimicen distintos parámetros de las composiciones resultantes, como el número de servicios implicados en la composición o la calidad de servicio global, además de una falta de análisis en términos de escalabilidad y rendimiento con un gran número de servicios.

Por otra parte, las técnicas centradas en datos se enfocan más en el problema de cómo extraer composiciones válidas mediante el análisis semántico de las dependencias entre las entradas y salidas de servicios relevantes. La mayor parte de estas técnicas se agrupan en: 1) técnicas clásicas de planificación automática, donde el problema de composición se traduce a un problema en el dominio de la planificación automática y se resuelve usando planificadores generales basados en espacio de estados, y 2) métodos basados en grafos que se encargan de construir grafos de dependencias de servicios mediante el análisis semántico de sus entradas y salidas para después aplicar algoritmos de búsqueda sobre el grafo para extraer composiciones válidas. Las técnicas clásicas de planificación automática se han venido usando en el campo de la composición automática de servicios para generar, mediante la traducción de servicios en acciones en el dominio de planificación, secuencias de servicios encadenados de manera que la ejecución secuencial transforme las entradas proporcionadas en las salidas esperadas, garantizando que se cumplen las precondiciones y efectos de cada uno de los servicios de la composición.

Estas técnicas funcionan bajo el supuesto de que los servicios son operadores complejos bien definidos en términos de entradas, salidas, precondiciones y efectos, y por lo tanto el problema de composición puede ser traducido de forma directa al dominio de planificación para ser resuelto con algoritmos generales de planificación. La mayor parte de la investigación en este tipo de técnicas aplicadas a la composición se han centrado en explotar técnicas semánticas y desarrollar mejores heurísticas para el rendimiento de los planificadores en el dominio de la composición. Como resultado, y en parte debido a la complejidad computacional del problema de composición con precondiciones y efectos, las propuestas existentes basadas en planificación no son capaces de generar soluciones óptimas en términos de número de servicios y/o calidad de servicio. Por contra, las técnicas basadas en grafos son por lo general más escalables pero a costa de sacrificar algunas características soportadas por los planificadores, como el manejo de precondiciones y efectos complejos. Este tipo de algoritmos suelen hacer uso de técnicas semánticas para buscar relaciones entre entradas y salidas de servicios, para a continuación aplicar distintos tipos de búsquedas para extraer composiciones optimizando distintos criterios, como tamaño de las soluciones o la calidad de servicio global.

Aunque este tipo de técnicas son por lo general capaces de generar soluciones buenas y de manera rápida, un análisis del estado del arte revela al menos dos problemas importantes y recurrentes: 1) En la mayoría de sistemas de composición, la noción de registro de servicios externo no existe, lo que implica que toda la información requerida se preprocesa y se carga en memoria principal antes de realizar la búsqueda. Esta es una importante limitación que impide el desarrollo de motores de composición que son capaces de trabajar con grandes conjuntos de datos y/o con registros distribuidos; y 2) las técnicas actuales fallan a la hora de generar soluciones óptimas minimizando eficazmente el número de servicios de las soluciones y/o la calidad de servicio global. Esto se debe principalmente a que, dada la complejidad del problema, la mayoría de las técnicas se centran sólo en la generación de soluciones rápidas pero subóptimas, en lugar de explorar la mejor forma de lograr un buen equilibrio entre velocidad y optimalidad.

El objetivo de esta tesis es hacer frente a estas limitaciones mediante el desarrollo de nuevas técnicas de composición de servicios que permitan generar composiciones óptimas minimizando el número de servicios de las soluciones así como la calidad de servicio global. Como parte de una investigación exploratoria inicial nos enfocamos en las técnicas centradas en control para generar composiciones complejas mediante la combinación de diferentes estructuras de control. Para ello, en el Capítulo 2 presentamos un algoritmo de programa-

ción genética que usa una gramática de contexto libre y un conjunto de operadores genéticos para generar *workflows* de composición válidos, minimizando el número de servicios y maximizando el paralelismo en la ejecución de los servicios seleccionados en la composición. Aunque los resultados obtenidos con esta técnica demuestran la eficacia de este enfoque en la generación de soluciones complejas usando las distintas estructuras de control definidas en la gramática formal, la complejidad del espacio de búsqueda y el tiempo de cálculo necesario para obtener soluciones buenas hacen que esta técnica no sea la más apropiada para la generación de soluciones *online*, pero sí para la generación y optimización *offline* de composiciones con estructuras de control complejas.

Para hacer frente a esta limitación, nos trasladamos hacia un enfoque centrado en los datos basado en la idea de construcción de grafos de dependencias de servicios mediante el análisis de las relaciones semánticas entre las entradas y salidas. Para ello, en el Capítulo 3 proponemos una serie de técnicas para generar, dada una petición de composición compuesta por una serie de entradas proporcionadas por el usuario y una serie de salidas esperadas, el grafo con los servicios relevantes y las relaciones semánticas entre ellos para dicha petición. Una vez generado el grafo, se aplican distintas optimizaciones para reducir su tamaño mediante la detección de servicios equivalentes y/o dominados en términos de funcionalidad. A continuación, se aplica un algoritmo basado en A* para extraer la composición con el menor número de servicio y la menor longitud del grafo. La validación con un conjunto de repositorios estándar del *Web Service Challenge 2008* demuestra que esta aproximación obtiene soluciones con un menor número de servicios que otras aproximaciones.

En el Capítulo 4, y en base a estos resultados previos, definimos un *framework* formal para composición basada en grafos que integra el descubrimiento de servicios basado en entradas y salidas como parte esencial en el proceso de composición, de forma que se puedan llevar a cabo composiciones de servicios sin necesidad de asumir disponibilidad local ni precarga en memoria del registro completo de servicios. En base a este *framework*, realizamos un análisis teórico de las implicaciones que tiene la integración entre el descubrimiento y la composición en términos de eficiencia y proponemos una serie de estrategias dirigidas a minimizar la sobrecarga del proceso de descubrimiento en la composición.

En el Capítulo 5 presentamos una extensión del anterior *framework* para incluir soporte para la optimización de la calidad de servicio global. Para ello, se extienden las optimizaciones presentadas en el capítulo anterior para detectar servicios equivalentes y dominados también en términos de calidad de servicio. Con el fin de extraer la composición óptima de este nuevo

grafo, minimizando tanto la calidad de servicio global como el número de servicios, se define un nuevo algoritmo híbrido que combina una búsqueda local rápida con una búsqueda global. La búsqueda local obtiene solución con un número de servicios subóptimo al tiempo que se satisface la calidad de servicio global de forma óptima. Por otra parte, la búsqueda global puede mejorar la solución obtenida con la estrategia de búsqueda local mediante la realización de una búsqueda exhaustiva combinatoria para seleccionar la composición con el menor número de servicios posible para la calidad de servicio óptima. Los resultados con los conjuntos de datos del *Web Service Challenge 2009-2010* y con un conjunto de datos generado aleatoriamente demuestran que con esta técnica es posible alcanzar un buen equilibrio entre optimalidad y velocidad.

La tesis termina con las conclusiones y el trabajo futuro presentados en el Capítulo 6.

Contents

CHAPTER 1

INTRODUCTION

1.1 Motivation

The growing importance of Service Oriented Computing (SOC) within the domain of distributed computing has led to an important increase in the number of available services both inside and outside of different organizations and companies worldwide. In this context, Web services have increased in popularity and have become established as the preferred technology for the development of distributed systems due to their interesting properties. Web services are self-described, loosely coupled software components that are network-accessible through standardized web protocols [78]. Each service exports certain functionality that can be invoked by passing the inputs and outputs required in the service signature. One of the key promises of services is the ability to compose and reuse their functionality to create new high-level components, a process known as *service composition*. Service composition leads to the creation of new services on-demand by combining the inputs and outputs of existing services in a way that fulfills some specific functionality that cannot be satisfied by a single service. Although the composition of services can be carried out by locating and designing compositions by hand, in a large-scale scenario it becomes necessary the use of automatic composition techniques to bring services to their full potential without the need of human intervention. However, automating the entire service composition process is a very hard and challenging task that requires not only the automation of other related activities, such as the *matchmaking* of input/outputs descriptions or the *discovery* of relevant services for the compositions, but also the optimization of different aspects that affect the solutions, such as the *size* of the resulting composition or the overall end-to-end *Quality-of-Service* (QoS). Partic-

ularly, optimizing both the size and the overall QoS of compositions is especially important in order to i) obtain small and manageable compositions that are easier to execute, debug and deploy and ii) to guarantee that the provided solutions are optimal in terms of response time, throughput or other non-functional quality attributes of the compositions. These problems have motivated researchers to approach the problem of automatic composition from different perspectives and applying different techniques [15, 61, 84, 92, 113, 114]. AI Planning techniques [61,84] have been traditionally used in service composition to generate valid composition plans by translating services into actions in the planning domain. These techniques work under the assumption that services are complex operators that are well defined in terms of inputs, outputs, preconditions and effects, so the problem can be directly solved using classical planning algorithms. However, given the complexity of generating satisfiable plans in the planning domain, these techniques are usually slow, do not generate neither optimal plans (in terms of minimizing the number of actions) nor optimal QoS-aware compositions, and present scalability issues in large repositories. Other approaches consider the composition problem as a data transformation process that can be described only in terms of inputs and outputs, leaving aside preconditions and effects. These approaches usually rely on graph-based techniques [41, 42, 44, 48, 54, 57, 75, 107, 120, 124, 125, 129] to efficiently extract compositions from a graph that contains both services and their input-output dependencies, generally using semantic techniques to enhance the automatic matching of inputs and outputs. Although these techniques are usually faster than other approaches, most techniques generate overly complex graphs with similar or redundant information which may negatively affect the overall performance and scalability in large service registries. Consequently, suboptimal search algorithms are used to explore only a subset of the space of possible solutions, and thus optimal solutions in terms of both the size and the overall QoS are not guaranteed.

But besides these current limitations, one of the assumptions that is also often made is that discovery and composition are two different and unrelated problems. As a result, most composition techniques do not deal with service discovery and assume instead that all the required services are locally available during the composition process. This unrealistic assumption requires preimporting all services locally which is only viable for those registries providing entire public dumps of the service descriptions they hold. Both discovery and composition are two interrelated activities that need to be addressed together in order to design better composition engines that can be integrated in real systems.

In this thesis we study the problem of generating automatic compositions of Web services,

optimizing both the size and the end-to-end QoS of the solutions and focusing on the semantic input-output parameter matching of services interfaces, i.e., given a composition request defined by a set of semantically annotated inputs and outputs, the goal is to select an optimal composite service whose invocation leads from the provided inputs to the expected outputs.

To this end, we devise an integrated graph-based framework that efficiently integrates the automatic service composition and semantic service discovery. The framework also includes a set of optimizations to improve the efficiency of the different steps involved in the whole composition and a set of algorithms that are able to efficiently extract optimal compositions in terms of both size and QoS. Our contributions have been validated with a comprehensive set of experiments using standard datasets, showing an improvement of the optimality of the solutions over state-of-the-art. Moreover, some of the techniques developed in this thesis are being currently used within the European COMPOSE project [88], coordinated by IBM Israel.

1.2 Web services

Service-Oriented Computing (SOC) has become the main paradigm for developing distributed applications [78]. In contrasts with the traditional monolithic paradigm, where applications are conceived as single processes managed by single organizations, SOC paradigm promotes the use of services as the main building blocks for the development of effective distributed applications that are easier to build, reuse and compose. Services are network-available, loosely-coupled software components that expose some business functionality through the use of standard interfaces [78]. Since services are self-contained and autonomous pieces of software, they can be easily described, published, located and invoked over a network, enabling the development of new applications by exploiting the collaboration and integration of applications across different organizations.

The requirements imposed by the SOC paradigm can be addressed in practice by the implementation of Service-Oriented Architectures. A Service-Oriented Architecture (SOA) is an architectural style that defines a set of principles and patterns for implementing service-based systems that fulfill an organization's business processes and goals [10]. A SOA is based on an interaction model which consists of three primary roles, represented in Figure 1.1. These roles are: *service providers*, responsible for providing the implementation of a service and publishing its description; *service clients*, that locate and request the execution of services; and *service registries*, which store information about the service capabilities,

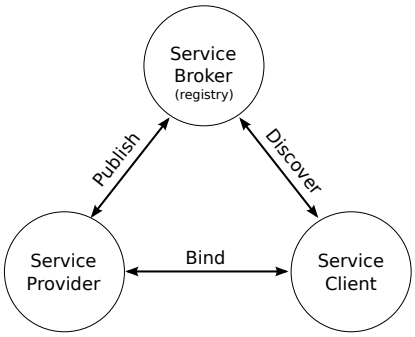

Figure 1.1: The three primary roles in Service Oriented Architectures.

interfaces, quality and other properties of the published services and provide a way to *discover* services for any potential service client.

Depending on their purpose, services can be categorized into two groups: *information-providing* services and *world-altering* services [64]:

- *Information-providing services.* These services are intended to: 1) provide some data (outputs), or 2) transform some *inputs* into some *outputs*. For example, a service offered by a thermal sensor only reads the information of the environment and returns a concrete value that represents the temperature at the time the service is queried. This service does not require any input since its main function is to read and retrieve information. Other services may require some inputs to operate. For example, a sentiment analysis service translates a text fragment (input) into a score associated to the positive or negative sentiment of the text (output). Information-providing services do not alter the state of the world, they only read data or transform data, but do not produce any side effect. As a result, their functionality can be described only in terms of *inputs* and *outputs*.

- *World-altering services.* The invocation of these services produces side effects that change the current state of the world. For example, a purchase service may use some inputs and produce some outputs, as in the case of information-providing services, but also produces some side effects: the number of available items changes after the invocation, as well as the money in the credit card of the client. In addition to the inputs and outputs, the functionality of these services is also described in terms of *preconditions*

(logical statements that must be satisfied in order to invoke the service) and *effects*[1] (logical statements that describe how the state of the world changes after the invocation of the service).

In this thesis we focus on *information-providing services* since: 1) are more lightweight; and 2) information-providing services greatly outnumber [105] world-altering services[2].

In the same way that services are the cornerstone of the SOC paradigm, *Web services* are the preferred way to implement services in SOAs. Web services are self-contained pieces of software that are implemented using a set of standard technologies to support inter-operable machine-to-machine interaction over a network [118]. The functionality of a Web service is described by a machine-readable interface description using WSDL (Web Service Description Language [29]), while the message interaction is specified through SOAP (Simple Object Access Protocol). WSDL abstractly describes the functionality of services in terms of operations, inputs and outputs, and provides mechanisms for binding the operations to concrete protocols and data format specification.

However, WSDL describes Web services only on the syntactic level, i.e., it does not provide the semantic information needed to declaratively define the functionality of Web services. This limitation hampers the automation of critical tasks such as the discovery, the composition or the invocation of services. Researchers have made many efforts to overcome this limitation by enriching service descriptions with semantic-based knowledge. As a result, many semantic description languages were proposed, such as SAWSDL [55], WSMO [102] or OWL-S [20], leading to the concept of Semantic Web Services. Semantic Web Services (SWS) were introduced by McIlraith et al. [64] as an extension of traditional services by annotating them with semantic descriptions in order to provide formal declarative definitions of their interfaces as well as to capture declaratively what the services do. On the basis of these semantic descriptions, SWS technologies introduce new powerful mechanisms to automate the tasks involved in the life cycle of service-oriented applications, for example by enabling the reasoning about inputs/outputs concepts to improve service discovery or to automate the chaining of multiple services by *matching* and connecting their inputs and outputs in order to create complex service compositions.

[1] Usually, the term *effect* from the planning domain is refered as *postcondition* in service modeling. We use both terms interchangeably.

[2] Here we consider that a service is a world-altering service if its preconditions and effects are explicitly modeled, regardless of whether its invocation produces some side effects or not.

Besides these syntactic and semantic *functional properties* of services, service descriptions can also include Quality-of-Service attributes (QoS). QoS refers to the *non-functional properties* that characterize the quality of the service, such as response time (the average time required to complete a web service request), throughput (the average number of web service requests served in a given time interval), availability (percentage of time that a service is available to consume), among others. These attributes provide a quantitative measurement of the performance and capabilities of services that are offered by different service providers. In order to support the QoS specification of Web services, a wide variety of alternatives have been proposed, that range from extensions to service registries [90] and service descriptions [35, 116] to more advanced mechanisms that enable the negotiation and monitor of QoS [31, 106]. These non-functional properties apply both to single services and to composite services. In a service composition, each individual service may affect the global end-to-end QoS of the composition. Thus, selecting the best combination of services with the appropriate QoS levels for a composition is a complex problem but fundamental to meet the customer's expectations.

In this thesis, we focus on the problem of generating optimal QoS-aware service compositions. For this purpose, we assume that the QoS attributes of a service are static values and are part of the service description itself.

1.3 Service Composition

One of the key SOC promises is the ability to rapidly deliver low-cost distributed applications by reusing and combining existing services [79]. This process of building new services by combining the functionality of many different services is called *service composition*. Although compositions can be manually created, this approach has important limitations:

- First, the manual design of service compositions is a very time-consuming task that requires to locate the appropriate services and to figure out the best way to combine them. Although this is feasible in some concrete domains with only few services, it is impractical when dealing with large repositories of services.
- Second, this approach requires to identify in advance the possible client needs and common use cases in order to design appropriate compositions that satisfy the demands. However, it is impossible to anticipate all users' needs in advance, even less to design, debug and deploy multiple compositions for every possible demand.

- Third, a failure in a single service of the composition can break the entire functionality of the composite service. This requires further manual labor to replace and redesign part of the composition, which means long downtimes that may negatively impact on business. It would be interesting to be able to recover from this unexpected situation by automatically generating an alternative composition that replaces the affected service or services in short time.

These limitations can be overcome by resorting to an automatic generation of Web service compositions. In this thesis we study the problem of generating automatic composition by finding the optimal way to combine the functionality of many information-providing services to generate *data-flow* compositions that satisfy a concrete goal. Informally, this problem can be formulated as follows:

> *Given a composition request expressed as a set of provided inputs and expected outputs, how can we quickly generate an optimal composition of services, minimizing both the number of services and the end-to-end QoS of the composition, that transforms the provided inputs into the expected outputs?*

The automatic composition of services comprises many challenges that range from how to discover relevant services to how to optimize the number of services or the overall QoS of the solution. Concretely, in this thesis we study this problem taking into account the following tasks that are strongly related with the generation of compositions:

- **I/O Matchmaking**. In a composition, the functionality of different services is combined by chaining the outputs produced by some services to the inputs required by other services. I/O matchmaking refers to the problem of computing an accurate match between inputs and outputs to generate valid compositions.

- **Service Discovery**. It is not possible to generate compositions if there is no mechanism for discovering relevant services. Service discovery is a fundamental task that needs to be carried out by any composition algorithm in order to locate adequate services, using the information available (inputs/outputs) at every step of the composition.

- **Optimal Service Composition**. This is the central problem of this thesis. The generation of optimal compositions comprises not only the efficient integration and execution of the previous tasks but also: 1) a formal model for computing the aggregated QoS of

a composition; 2) the design of efficient algorithms for the generation of optimal compositions, minimizing the size and optimizing the overall QoS of the composition, and 3) the development of optimization techniques to improve the scalability of the whole composition system.

In the following subsections, we analyze the available state-of-the-art methods and techniques as well as the main assumptions that are made in this thesis.

1.3.1 I/O Matchmaking

A fundamental issue that needs to be addressed for generating compositions and even for discovering services, is the ability to analyse the compatibility between inputs types and outputs types. Basically, in a composition, the functionality of many services is combined by connecting (*matching*) the outputs of a service with the inputs of other service, generating a complex data-flow of inputs and outputs. The correct behavior of the entire composite service depends in part on the correctness of the I/O matching process, i.e., an I/O mismatch can cause the malfunction of an entire composition. Automating this process while ensuring correctness can be very hard to accomplish using only the syntactic description of the services' functionality, since potentially compatible inputs and outputs can differ due to syntactic differences in their definitions [76]. For example, suppose that a `ContactInfoService` is a service that given the name of a worker, returns information such as her phone and the information of the building where she is located in. On the other hand, a service `LocationService` takes a location and returns the geographical coordinates of that location. If we want to obtain the geolocation of a worker, we could compose both services, and so the output `Building` of the first service can be passed as a `Location` to the `LocationService` to obtain the geolocation of the building. However, it is impossible for a syntactic-based system to realize that both terms are *semantically* related. Clearly, using syntactic descriptions of the inputs and outputs is insufficient to detect these potentially valid matches.

To enhance the automatic I/O matchmaking, inputs and outputs of services can be annotated with semantic concepts from an *ontology* [37]. An ontology models a common and structured vocabulary which defines the concepts and their logic relationships used to describe a concrete domain. By using semantic annotations, the I/O matching process can perform logical inferences to derive additional information about the concepts of the ontology, leading to the recognition of valid input-output matches despite their syntactic differences. This func-

tionality, which we refer to as I/O semantic matchmaking, is in charge of assessing the level of semantic compatibility between concepts. To do so, semantic matchmaking relies on semantic reasoning in order to be able to determine the relationships between the concepts.

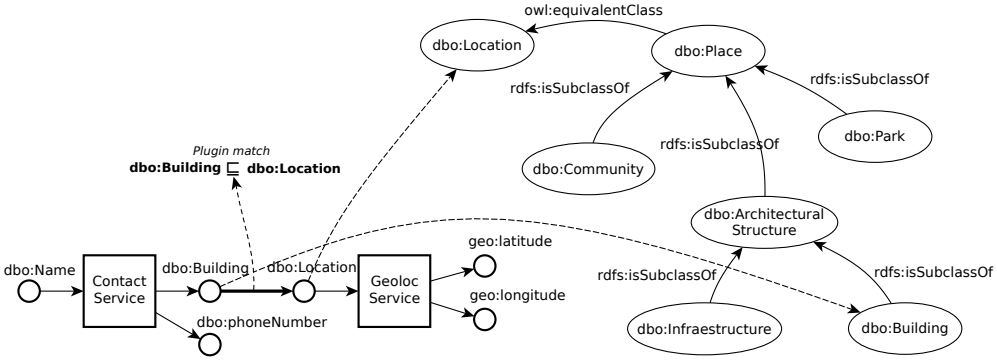

Figure 1.2: I/O Matchmaking in a composition.

Figure 1.2 shows the same example as before, using semantic concepts to annotate inputs and outputs. The ontology is depicted on the right of the figure. As it is shown, by exploring the relations in the ontology, it is possible to see that a `Building` is a type of `ArchitecturalStructure`, which in turn is a type of `Place`. Also, since a `Place` is equivalent to a `Location`, we can conclude that a `Building` is a type of `Location`. This reasoning process allows to derive facts that are not explicitly expressed in the ontology. Depending on their relationship, concepts can match with different qualities (or matching degrees). The different matching degrees that are typically contemplated in the literature are [76]: *exact*, if both concepts are equivalent; *plug-in*, if a concept c_1 is more specific than a concept c_2; *subsume*, if a concept c_1 is more general than a concept c_2; and *fail* if both concepts are not related to each other.

This reasoning process can be used for example, to discover services that can consume or produce a concrete input/output by finding semantically compatible types, as well as to find out which services can be used to pass some information to other services. *In this thesis, we use this type of subsumption reasoning as a tool to improve the I/O matchmaking process, by assuming that inputs and outputs of services are correctly annotated with concepts from ontologies.*

1.3.2 Service Discovery

Service discovery has traditionally been considered as the problem of finding the services that best match a given request in terms of its functional behavior and user preferences [64]. As a consequence the interface exposed by discovery engines assumes that requests are fully specified in terms of a well-defined interface and categorization, i.e., discovery systems expect a precise description of the service in terms of inputs and outputs, and/or other characteristics such as preconditions and effects (*service signature*). Research efforts in this area have led to the development of powerful (though generally slow) *service matchmakers* [52, 59].

However, one of the reasons for which service composition is required in first place is because there is usually no single service that can fully match a request, but it could be matched by the composition of many individual services instead. In fact, during automatic composition an exploratory search is usually required to guess which relevant services can be selected using incomplete and partial information. Figure 1.3 shows an example of these differences. A typical service discovery is represented on the left, where the goal is to find all those services that fully satisfy the discovery request (all those services that consume `in1` and `in2` and produce `out1` and `out2` in this case). In the figure on the right, there is no single service that can consume and produce the whole set of inputs and outputs, so a suitable combination of services should be found instead. In this case, the discovery is performed using the information provided by the user and the information generated by other candidate services. This requires to launch many simple requests (or fine-grained queries), rather than fully specified requests, in order to locate relevant services that match some partial information available to the algorithm (e.g., services that consume some inputs and/or produce some outputs). Hence, response time of discovery systems becomes a crucial issue to be addressed in order to design better discovery engines that can seamlessly be integrated within the whole discovery process without a great impact on the overall response time.

However, most of the work has been focused on improving the retrieval performance (i.e., precision-recall curve) without much concern about the response time requirements and/or the API requirements to provide an efficient input/output fine-grained discovery granularity for automatic composition. As a result, response times of discovery engines are orders of magnitude above what would be acceptable for a composition engine that should delegate the thousands of discovery requests it needs to issue at composition time, although this aspect is recently gaining in importance [49].

A direct consequence of this is that most composition engines re-implement locally their

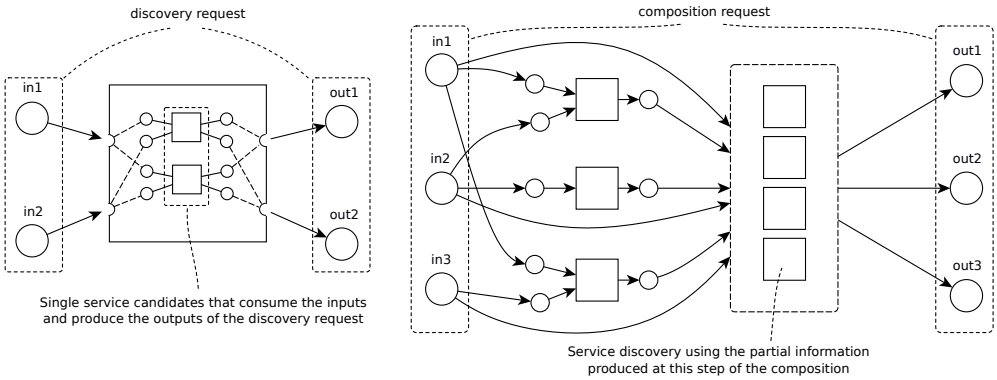

Figure 1.3: Traditional service discovery (left figure) where only single candidate services that fully match the inputs and outputs of the request are retrieved vs. service discovery in a composition using partial information (right figure), where services are discovered using the information (inputs, outputs) available at the current step of the composition.

own discovery methods. For example, in [54], the authors present an efficient framework for Web service composition that supports semantic Web service discovery, but the notion of an external service registry is missing, so all the information required is preprocessed and loaded in the main memory. This assumption requires preimporting all services locally which is only viable for those registries providing entire public dumps of the service descriptions they hold. This is a recurrent issue shared by many other composition systems [3, 44, 68, 70, 89, 107, 129]. There is remarkably little research about: i) how to efficiently integrate semantic discovery and composition, ii) how this integration affects the performance of the overall composition, and iii) how the overhead of the discovery can be minimized in order to obtain optimal compositions faster. Some interesting frameworks that partially address the first issue are [34, 57]. In [57] Lécué et al. develop an integrated framework for dynamic Web service composition that exploits the semantic matchmaking to discover relevant services. However, the discovery is performed at a very high level, assuming that services have a well-defined semantic goal description of their functionality instead of directly exploiting the input/output information of the services, and it is used to find all the possible candidate services before the composition takes place. One of the problems is that it is not always possible to detect every potential service beforehand using a very high level description of the goals. This can prevent the selection of services whose functionality indirectly contributes towards the goals,

for example by translating some data between services, or performing authentication required by some service in the composition. Also, there is a lack in terms of how the discovery process affects the response time of the whole composition. Similarly, in [34], the authors present a composition framework that supports both automatic semantic discovery and composition, among other relevant phases of the composition life-cycle, such as service publication and service selection, taking into account non-functional properties. Again, since the discovery is not interleaved with the composition phase, this approach shares the same problems as the previous work. Although in this work the authors offer a more detailed evaluation of the response time of both the discovery and the composition, they do not discuss any strategy to reduce the overhead of the discovery phase.

Other works, such as [11] and [19], offer an interesting alternative view of the service discovery. Instead of considering the discovery as a key part of the whole composition process, the discovery is viewed as a single process that is able to not only discover single services but also composite services when there is no single service that fully matches the request, so there is no clear distinction between discovery and composition. However, this lack of clear separation between discovery and composition may hamper the development of better composition systems by integrating the latest advances in both fields, and can also prevent the study of the relation of these tasks in order to build better systems. Furthermore, the lack of experimental validation in these works makes it hard to judge the real effectiveness and the advantages of the automatic discovery by means of composition.

All these current limitations prevent the development of faster composition systems where discovery and composition are two interrelated activities. *In this thesis, we consider the discovery of services as an interrelated task that is interleaved with the composition task. To do so, we clearly separate both activities through a simple fine-grained API to discover services based on the semantic information of their inputs and outputs. We also analyze the impact that this implication has in terms of response time. Moreover, we provide a reference implementation based on the integration of discovery and composition to validate this approach, and we discuss different mechanisms to minimize the overhead of the discovery process in the whole composition process.*

1.3.3 Optimal Service Composition

Optimal service composition refers to the generation of composite services optimizing one or more different properties such as the size or the end-to-end QoS. However, due to the large

amount of available services, the generation of optimal compositions is a very complex and broad task that comprises many different kinds of problems, and so there is no an unique category of techniques. As a result, many different approaches have been proposed, each one focusing on different aspects of the composition problem and under different assumptions [15, 61, 84, 92, 113, 114]. According to [1], automatic composition techniques can be broadly classified into *template-based* and *search-based* depending on whether the composition workflow is manually generated (where a template or a set of possible templates are provided to the composition system) or automatically generated from scratch. A *workflow*, in the context of service composition, defines a set of tasks and their control and data dependencies, where each task in the workflow specifies an abstract functionality that can be implemented by many different concrete services.

In ***template-based*** compositions, a template of the composition workflow is defined in advance by an expert, who is in charge of defining the different abstract tasks and modeling their control and data dependencies in advance. Once the template is designed, the composition system is responsible of instantiating each task in the template in order to obtain an executable workflow. There are usually two main problems these approaches focus on: 1) how to efficiently discover and match services for each task and 2) how to optimize the end-to-end QoS for a fixed set of service candidates. The former problem is more focused on efficient discovery and matching of services using semantic techniques [2, 23, 65, 109] whereas in the latter problem, also referred as QoS-aware service selection [131, 132], services candidates are usually in place (there is no explicit discovery) and the goal is to select the best service (from the fixed set of candidates) for each task in order to locally or globally optimize the overall QoS of the composition workflow [8, 21, 119, 133].

Although these approaches are very flexible and can handle complex user requirements, template-based approaches have some important limitations. On one hand, since the composition workflow is predefined, the number of services is also tied to the number of abstract tasks in the workflow, making impossible the optimization of the composition size. Thus, if an abstract task in the workflow has no candidate services, it is not possible to generate an alternative composition workflow that could replace the non-instantiable task with a different but functionally equivalent combination of services for that task. On the other hand, there can be situations where certain QoS values are missing or cannot be measured, and so optimizing only the overall QoS may be not sufficient to obtain good solutions. In this context, optimizing not only the available QoS but also the number of services of the composition may

indirectly improve other missing QoS properties.

In ***search-based*** compositions, the composition process includes the automatic generation of the entire composition workflow, enabling the construction of compositions with variable size and structure. Certainly, these kind of automatic service compositions are more powerful but computationally harder since, in general, there is an exponential number of ways in which different services can be combined to accomplish the same task. Moreover, the generation of automatic composition workflows requires not only the generation of the control-flow but also the correct information exchanging (data-flow) through the interaction of inputs and outputs between services.

Considering this distinction, there are at least two ways to approach the automatic composition problem depending on the strategy used to generate the workflows: a *control-centric* and a *data-centric* approach [67]. The control-centric approach combines different control constructions that are used to coordinate the control-flow of the composition, and then places suitable services for each task, whereas the data-centric approach analyzes the data dependencies between the inputs and outputs of services (i.e., which outputs can be potentially used to pass as inputs of other services) and then infers the control dependencies from the data once the data dependencies are resolved. Control-centric approaches are usually computationally harder since the use of different control constructions increments the variability of the solutions as well as the number of similar compositions to accomplish the same goal. This increased complexity works against obtaining reasonable good solutions in short time. In contrast, data-centric approaches are simpler to model and to validate but at the cost of less expressiveness since not all the possible control constructions can be inferred just by analyzing data dependencies. Each strategy has its own advantages and limitations, and its effectiveness would depend on the particular requirements and the specific context in which the composition will take place.

Most of the control-centric techniques for automatic generation of composition workflows are mainly based on evolutionary algorithms [12,26,117,127] and Hierarchical-Task-Network planners [27, 110, 112]. Evolutionary techniques, and concretely Genetic Programming (GP) algorithms, are well suited since GP offers good performance in combinatorial optimization problems with large search spaces. GP is a type of Evolutionary Algorithm (EA) that stochastically generates and evolves a population of individuals (solutions) via natural selection. The information of each individual is represented by its genotype: a collection of genes that encode the characteristics (or phenotype) of the individuals. The main difference with other

EAs is that GP uses tree structures to represent the genotype of the individuals, as well as tree-based operators to transform the genotype during the optimization process. These tree-like structures used to encode the instances of the problem bear much resemblance with the way in which control-centric compositions of services are defined, i.e., by means of a hierarchical, recursive combination of control constructions. Similarly, HTNs use a hierarchical top-down decomposition of tasks to automatically generate plans (workflows) but in a completely different manner. Instead of evolving a set of randomly generated workflows, HTNs apply domain-specific operators to decompose high-level descriptions of the composition goal into fine-grained tasks that are directly implemented by Web services. HTNs share some similarities with template-based methods since a description of the different tasks and decompositions have to be modeled in advance by an expert. But regardless of the strategy adopted, there is a lack of works that analyze how these workflows can be optimized, i.e., how to generate optimized workflows maximizing parallelism and reducing the number of services for a concrete problem. Furthermore, there is also a lack of analysis in terms of scalability and performance in large-scale scenarios.

On the other hand, data-centric techniques are focused on how to extract compositions by inspecting semantic dependencies between inputs and outputs of candidate services that are potentially eligibles for the final composition. Most approaches can be categorized into: 1) classical AI planning approaches, where the composition problem is translated into the planning domain and solved using general planners [61, 84], and 2) graph-based approaches that build a graph with the services and their input/output semantic relations (usually ignoring preconditions and effects), and apply graph search techniques to extract service compositions from the graph [41, 42, 44, 48, 54, 57, 75, 107, 120, 124, 125, 129]. AI Planning techniques have been traditionally used in service composition to generate valid composition plans by mapping services to actions in the planning domain. These techniques work under the assumption that services are complex operators that are well defined in terms of inputs, outputs, preconditions and effects, so the problem can be translated to a planning problem and solved using classical planning algorithms. Most of these approaches have been mainly focused on exploiting semantic techniques [5, 43, 110] and developing heuristics [5, 51, 70] to improve the performance of the planners. As a result, and partly due to the complexity of generating satisfiable plans in the planning domain, these approaches do not generate neither optimal plans (minimizing the number of actions) nor optimal QoS-aware service compositions. On contrast, graph-based composition approaches are usually more scalable [72], but at the ex-

pense of ruling out some of the features supported by AI planners, such as preconditions and effects. These approaches exploit the semantic relations between inputs and outputs of services in order to generate graphs of related services. Then, different optimizations and search algorithms can be applied on the graph to extract valid compositions, optimizing different criteria such as complexity of the solutions or QoS. Although these approaches show generally good performance and low response times, we identified two important recurrent problems:

1. In most composition systems, the notion of an external service registry is missing, which implies that all the information required is preprocessed and loaded in the main memory. This is an important limitation that prevents the development of composition engines that are able to cope with large and/or distributed datasets.

2. Current techniques fail to generate optimal solutions by effectively *minimizing the number of services* in large datasets. This is mainly because, given the complexity of the problem, most approaches focus only on generating fast, suboptimal solutions instead of exploring how to achieve a good tradeoff between speed and optimality. This has led to a lack of composition engines based on efficient algorithms and optimization techniques that can generate optimal compositions in terms of number of services but also in terms of the end-to-end QoS of the solutions.

Concretely, the second problem is very interesting and important, since optimizing the composition size can bring important benefits to the different roles in a service scenario, namely: brokers, service providers and clients. From the point of view of a broker, the generation of smaller compositions is interesting to achieve manageable compositions that are easier to debug, execute, monitor, deploy and scale. Clients can also benefit from smaller compositions, especially when there are multiple solutions with the same optimal end-to-end QoS but different number of services. This is the case when, for example, a concrete functionality cannot be achieved without consuming some critical service that acts as a bottleneck by limiting the optimal end-to-end QoS achievable and for which there is no possible replacement. In scenarios like this, it is better to return the smallest composition among the whole set of possible valid compositions that share the same end-to-end QoS, since decreasing the number of services involved in the composition may indirectly improve other quality parameters such as communication overhead, risk of failure, connection latency, etc. Minimizing the number of services of a composition is also interesting from the perspective of service providers. For

example, if the client wants the cheapest composition, the solution with fewer services from the same provider may also require less resources for the same task.

However, despite the clear benefits that the service minimization can bring to the different parties involved in a service composition scenario, remarkably little research has been done so far. Most of the work focused on minimizing also the size of the compositions started ganing momentum since the appearance of the Web Service Challenge [13]. This challenge motivated the development of new algorithms for service composition, mostly graph-based approaches, that introduce some ideas inspired by different fields, such as AI Planning, Heuristic Search or Operations Research. Most notable works are the top-3 winners of the WSC'08 [3,70,129] and the winners of the WSC'09-10 [4,46,130]. Although these approaches show generally good performance with low response times, they cannot guarantee to obtain optimal solutions in terms of both number of services and end-to-end QoS. Additionally, none of these systems consider neither the integration with service registries nor the use of service optimizations to deal with potential scalability problems.

The work presented in this thesis contributes towards developing better techniques to cope with these issues. To this end, we explore the limitations of the current control-centric and data-centric approaches on large datasets and we propose new methods aimed to achieve better tradeoffs between speed and optimality of the solutions in terms of number of services and QoS.

1.4 Research Contributions

The aim of this thesis is to advance the state-of-the-art in the field of Web service composition, focusing on *information-providing* services, by developing new algorithms and techniques to cope with the limitations presented in the previous section. As part of a first exploratory research, we first developed a novel control-centric approach based on a genetic algorithm to generate composition workflows minimizing the number of services and maximizing the parallel execution of services (*C1*). Despite demonstrating the effectiveness of this approach to deal with the extremely large search space of all the possible combinations of services and control constructions, the high computation time required to solve some particularly large instances of the problem prevents its application for generating "on the fly" compositions at run time. To deal with this difficulty, we moved towards a more lightweight data-centric approach based on the generation of I/O matching graphs (*C2*). We developed a set of algorithms and

optimization techniques to generate optimized graphs and to select the best combination of services from the graph, minimizing the number of services of the solutions and the end-to-end QoS (*C4*). All the developed methods are collected into an integrated graph-based framework for service composition that supports I/O based service discovery from external registries (*C3 and C4*). More concretely, the contributions of this thesis are:

C1. Composition of Web Services through Genetic Programming

We developed a genetic programming algorithm that automatically generates composition workflows of services. The algorithm uses a context-free grammar to limit the valid control structures, takes into account the attributes updating, and minimizes both the number of services of the composite solution and the workflow length or execution path needed to achieve the desired result.

C2. An Optimal and Complete Algorithm for Automatic Web Service Composition

We addressed the problem of the web service composition as a graph search problem from the point of view of the semantic input-output message structure matching, i.e, we did not take into consideration the non-functional properties. The contributions are: (1) the method is able to calculate, given a request, an extended service dependency graph which represents a valid but sub-optimal solution for the request; (2) the heuristic search algorithm, based on the well-known A*, finds all optimal solutions from the point of view of the number of services and execution path, maximizing the parallel execution of services and minimizing the total number of services; 3) we define a set of optimizations to reduce the graph size, based on the redundancy analysis and service dominance; and 4) we include a method to dynamically reduce the possible paths to explore during the search by filtering equivalent compositions.

C3. An Integrated Semantic Web Service Discovery and Composition Framework

Based on our previous research, we defined a formal graph-based framework focused on the semantic input-output parameter matching of services' interfaces that efficiently integrates the automatic service composition and semantic service discovery. The contributions are: (1) a formal framework that presents a theoretical analysis of graph-based service composition in terms of its dependency with a service discovery, and we provide a fine-grained I/O discovery interface which reduces the performance overhead without having to assume the local availability and in-memory preloading of service registries.

The framework also includes an optimal composition search algorithm to extract the best composition from the graph minimizing the length and the number of services, and different graph optimizations to improve the scalability of the system, which as far as we now are not included in other frameworks; (2) a reference implementation of this formal framework based on the adaptation of two independently developed components, namely ComposIT and iServe, respectively in charge of service composition and discovery; (3) a detailed performance analysis of the integrated system, highlighting both the unacceptable performance achieved when using the typical out of the box discovery implementations, as well as the fact that top performance is achievable with the adequate discovery granularity and corresponding indexing optimizations.

C4. Hybrid Optimization Algorithm for Large-Scale QoS-Aware Service Composition

We extended the graph-based framework to incorporate Quality-of-Service aspects and service minimization. The contributions are: (1) a multi-step optimization pipeline based on the analysis of non-relevant, equivalent and dominated services in terms of interface functionality and QoS; (2) a fast local search strategy that guarantees to obtain a near-optimal number of services while satisfying the optimal end-to-end QoS for an input-output based composition request; and (3) an optimal combinatorial search that can improve the solution obtained with the local search strategy by performing an exhaustive combinatorial search to select the composition with the minimum number of services for the optimal QoS.

1.5 Publications

The following publications are the result of this PhD thesis:

- **Journal Papers**:
  - Pablo Rodríguez-Mier, Manuel Mucientes, and Manuel Lama. Hybrid Optimization Algorithm for Large-Scale QoS-Aware Service Composition. *IEEE Transactions on Services Computing*, 2015 (DOI 10.1109/TSC.2015.2480396).
  - Pablo Rodríguez-Mier, Carlos Pedrinaci, Manuel Lama, and Manuel Mucientes. An Integrated Semantic Web Service Discovery and Composition Framework. *IEEE Transactions on Services Computing*, 2015 (DOI 10.1109/TSC.2015.2402679).

  - Pablo Rodríguez-Mier, Manuel Mucientes, Juan Carlos Vidal, and Manuel Lama. An Optimal and Complete Algorithm for Automatic Web Service Composition. *International Journal of Web Service Research*, 9(2):1–20, 2012.
  - Pablo Rodríguez-Mier, Manuel Mucientes, Manuel Lama, and Miguel I. Couto. Composition of web services through genetic programming. *Evolutionary Intelligence*, 3(3-4):171–186, 2010.

- **Conferences**:
  - International conferences:
    * Pablo Rodríguez-Mier, Manuel Mucientes, and Manuel Lama. A Hybrid Local-Global Optimization Strategy for QoS-aware Service Composition. In *Proceedings of the 22nd IEEE International Conference on Web Services (ICWS)*, pages 735-738, 2015.
    * Pablo Rodríguez-Mier, Adrián González-Sieira, Manuel Mucientes, Manuel Lama, and Alberto Bugarín. Hipster: An Open Source Java Library for Heuristic Search. In *Proceedings of the 9th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 481-486, 2014.
    * Pablo Rodríguez-Mier, Manuel Mucientes, and Manuel Lama. A Dynamic QoS-Aware Semantic Web Service Composition Algorithm. In *Proceedings of the 10th International Conference on Service-Oriented Computing (ICSOC)*, pages 623-630, 2012.
    * Pablo Rodríguez-Mier, Manuel Mucientes, and Manuel Lama. Automatic Web Service Composition with a Heuristic-Based Search Algorithm. In *Proceedings of IEEE 9th International Conference on Web Services (ICWS)*, pages 81–88, 2011.
  - National conferences:
    * Pablo Rodríguez-Mier, Manuel Mucientes, and Manuel Lama. Algoritmo Híbrido de Composición Automática de Servicios con QoS. In *Jornadas de Ciencia e Ingeniería de Servicios (JCIS)*, 2015.
    * Pablo Rodríguez-Mier, Manuel Mucientes, and Manuel Lama. An Optimal and Fast Algorithm for Web Service Composition. In *Jornadas de Ciencia e Ingeniería de Servicios (JCIS)*, pages 3–8, 2011.

1.6 Thesis Outline

This dissertation is organized as follows:

- Chapter 2 presents a Genetic Algorithm that uses a formal context-free grammar to generate valid populations of composite services using the different control structures defined in the grammar. The algorithm makes use of different operators to evolve the initial population of solutions and different techniques to reduce the complexity of the workflows during the search. The goal of the algorithm is to optimize a fitness function that takes into account both the number of services in the workflow and the execution path. In order to validate the performance of the approach, we present a detailed evaluation using different datasets with up to 1,090 services.

- Chapter 3 focuses on the composition problem from a data-centric perspective that is better suited to generate compositions faster. For this purpose, we develop a graph-based algorithm that analyzes the semantic information of the services to generate a graph that contains all the relevant services for the composition. Once the graph is generated, we apply different optimizations to reduce its size by detecting services that are equivalent or dominated in terms of their functional interface. Then, an A*-based algorithm is applied to find the optimal composition from the graph, minimizing the total number of services and the length of the solution, and we provide a comprehensive validation of the algorithm with the standard datasets of the Web Service Challenge 2008.

- Chapter 4 presents a graph-based framework that integrates both service discovery and optimal service composition. This formal framework provides a theoretical analysis of graph-based service composition in terms of its dependency with a service discovery without assuming the local availability and in-memory preloading of service registries. We also provide a reference implementation of this formal framework based on the adaptation of two independently developed components, that is used to empirically study the impact of the discovery task in the whole composition using different optimization mechanisms under different conditions.

- Chapter 5 presents an extension of the previous framework to include support for both service minimization and end-to-end QoS optimization. We extend the optimizations presented in the previous chapter to take into account QoS and we introduce a new step

in the optimization pipeline to prune suboptimal QoS services. In order to extract the optimal composition from this new extended graph, minimizing both the end-to-end QoS and the number of services, we develop a novel hybrid local-global search algorithm that combines a fast local search with a global search. The local search obtains a near-optimal number of services while satisfying the optimal end-to-end QoS. On the other hand, the global search can improve the solution obtained with the local search strategy by performing an exhaustive combinatorial search to select the composition with the minimum number of services for the optimal QoS. A comprehensive validation with the datasets of the Web Service Challenge 2009 and with random generated datasets is also provided.

- Chapter 6 presents the main conclusions and the future work.

Chapter 2

Composition of Web Services through Genetic Programming

Control-centric approaches for service composition focus on the automatic generation of complex composition workflows by combining different control structures such as sequences, choices, splits or loops, among others, that are used to coordinate the control-flow of the services in the workflow. One of the limitations of current approaches, as commented in Chapter 1, is the lack of works that analyze how to generate optimized composition workflows maximizing the paralellism and minimizing the number of services used within the workflow, in order to avoid the generation of overly complex and inefficient compositions. Furthermore, there is also a lack of analysis in terms of scalability and performance using standard datasets. As part of a first exploratory research made in this thesis, in this work we develop a Genetic Algorithm that uses a formal context-free grammar to generate valid populations of composite services using the different control structures defined in the grammar. The algorithm uses different genetic operators to evolve the initial population of solutions and different techniques to reduce the complexity of the workflows during the search. The goal of the algorithm is to optimize a fitness function that takes into account both the number of services in the workflow and the execution path. In order to validate the performance of the approach, we present a detailed evaluation using different datasets with up to 1,090 services.

This chapter includes a full copy of the following journal paper that describes in detail the proposed approach:

Pablo Rodríguez-Mier[1], Manuel Mucientes[1], Manuel Lama[1], and Miguel I. Couto[1]. Composition of web services through genetic programming. *Evolutionary Intelligence*, 3(3-4):171–186, 2010. Springer. ISSN: 1864-5909. DOI: 10.1007/s12065-010-0042-z. URL: `http://dx.doi.org/10.1007/s12065-010-0042-z`.

2.1 Abstract

Web Services are interfaces that describe a collection of operations that are network-accessible through standardized web protocols. When a required operation is not found, several services can be compounded to get a composite service that performs the desired task. To find this composite service a search process in a, generally, huge search space must be performed. The algorithm that composes the services must select the adequate atomic processes and, also, must choose the correct way to combine them using the different available control structures. In this paper a genetic programming algorithm for web services composition is presented. The algorithm has a context-free grammar to generate the valid structures of the composite services and, also, it includes a method to update the attributes of each node. Moreover, the proposal tries to minimize the number of services, and looks for compositions with the minimum execution path. A full experimental validation with four different repositories with up to 1,090 web services has been done, showing a great performance in all the tests as the algorithm finds a valid solution with a short execution path.

2.2 Introduction

Web Services are interfaces that describe a collection of operations that are network-accessible through standardized web protocols, and whose features are described using a standard XML-based language [7,33]. This includes functional features that indicate the input/output needed to invoke the execution of a web service; nonfunctional features such as cost, robustness, reliability, etc.; interaction features or choreography that describe how a client dialogs with the service in order to consume its functionality; and structural features or orchestration that model how the internal components of the service are combined to execute it.

[1]Centro Singular de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela.

In this way, as the characteristics are available through the interfaces, web services can be automatically discovered and invoked by extern programs (clients). When programs do not find a service with the required functionality (inputs and outputs), it is possible to compose a new service automatically. This composite service combines the functionalities of other services to get the desired outputs. This combination consists of a set of services that are executed in a sequence or in a set of workflow-like structures that control the execution of the services (specified through web services composition languages as OWL-S [62] or BPEL4WS [32]).

In the last years several papers have dealt with the composition of web services. Some approaches consider the composition problem as a planning problem of several actions (services) that operate on an initial state (inputs and preconditions) and generate an output state (postconditions) [45, 50, 73, 86, 91, 110, 126]. In these proposals, the planning techniques are blended with semantic reasoning to combine the outputs of some services with the inputs of others. The main drawback is that in these approaches the result of the composition is a sequence of services and, therefore, they do not take into account other control constructions that are part of the OWL-S or BPEL4WS models. In this way, this particular problem has a computational complexity much lower that those compositions that follow languages like OWL-S or BPEL4WS.

Other papers solve the composition of services with machine learning techniques like genetic programming [12, 26, 117, 127]. In these approaches, the minimum execution path needed to achieve a solution is not considered in the fitness function, and therefore optimal individuals are not assured. Furthermore, these proposals are validated with a low number of services and then the effectiveness of the proposed algorithms cannot be really evaluated.

In this paper we present a genetic programming algorithm that solves the problem of composition of web services. The algorithm uses a context-free grammar to limit the valid structures, takes into account the attributes updating, and minimizes both the number of services of the composite solution and the execution path needed to achieve the desired result. A full validation has been done in four different repositories: OWL-S TC [56], a hand-made repository with 1,000 services, and three program-generated repositories proposed for the 2008 Web Service Challenge of the EEE conference [13]. The behavior of the algorithm shows a great performance, as in all the cases a correct composition was found. This validation demonstrates the generality of the evolutionary algorithm, as it does not depend on the structure and features of a given repository.

The paper is structured as follows: Sec. 2.3 introduces the web services composition problem, and Sec. 2.4 describes the different approaches that have already been proposed. Then, Sec. 2.5 presents the proposed genetic programming-based algorithm for web services composition, Sec. 2.6 comments the obtained results and, finally, Sec. 2.7 points out the conclusions.

2.3 Problem Description

In this paper, we consider that web services are only characterized by their functional features (that is, inputs and outputs), which are semantically described through ontologies. With this semantic description the output of a service O_{So} matches the input of other service I_{Si} when O_{So} is a subclass of I_{Si}. In general, when a concept C_i is a subclass of a concept C_j ($C_i \subseteq C_j$), then there is a semantic matching between C_i and C_j. This semantic matching will be used when two o more concepts are compared in the different stages of our algorithm.

Considering this description for web services, the composition problem can be formulated as the automatic construction of a workflow that coordinates the execution of a set of services that interact among them through their inputs and outputs (applying the semantic matching). This workflow, therefore, has services and a set of control structures that define both the behavior of the execution flow and the inputs/outputs of the services related to those structures. Thus typical control structures of web services composition languages are:

- *Sequence structure*, where the output of a service is the input of one of the following services of the sequence. This is the simplest control structure of the workflow languages.

- *Selection (choice) structure*, where an output can be achieved through two or more services, which therefore share the same output, but only one service will be selected and executed.

- *Parallel (split) structure*, where two o more services are executed in parallel and, as result, produce several and different outputs.

- *Parallel and synchronized (splitJoin) structure*, where the execution ending of services that run in parallel is synchronized. In this construction the services outputs are tipically different.

- *Loop structure*, where a set of services are executed until a given condition is verified. This structure does not impose any condition to the input/output concepts, although some approaches [12] assume that there must be a set of data in order to be individually used in each loop iteration.

These structures are shared by OWL-S[2] and BPEL4WS, which are the languages of the service repositories we have used to validate the proposed algorithm. In this sense it is important emphasize that the proposed algorithm is *independent* of the web services composition language, because the behavior of the control structures is defined in a general way.

As has been mentioned, the composition of web services consists of a set of services that are executed in a sequence or in a set of workflow-like structures that control the execution of the services. These two problems are very different from the point of view of the computational complexity:

- Sequence-based compositions: the complexity is $O(n!)$, where n is the size of the services repository[3].

- Workflow-like structures: the complexity is $O(n!\, t)$, where t is the number of different structures that can be generated. The structures can be represented by trees, and can be defined by a context-free grammar. Therefore, t depends on the grammar and on the maximum tree depth (d). If we assume that the grammar generates a complete binary tree[4], the maximum number of leaf nodes is 2^d. Thus, if the grammar has m different control structures, then the number of different structures that can be generated is $t \propto m^{2^d}$, as for each leaf node a different control structure can be selected. t is proportional to this expression because not all the rules in the grammar generate two internal nodes and/or not all the leaf nodes are control structures. This is the case for the context-free grammar defined in this paper (described in section Sec. 2.5.1, Fig. 2.2). Finally, the complexity of workflow-like structures is $O\left(n!\, m^{2^d}\right)$.

[2] In OWL-S a *process* is used with the same meaning as a service. Thus a single service is named as an atomic process, and composite services are named as composite processes. We use this notation in the grammar that describes the chromosomes of our evolutionary algorithm.

[3] This complexity is for the worst case: a composition which uses all the services of the repository. However, if we knew in advance the size of the composition (this is, in general, not truth), the complexity would be $O\left(\frac{n!}{(n-p)!}\right)$, where p is the number of services of the composition.

[4] In a complete binary tree every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

As can be seen, workflow-like compositions have a much higher number of candidate solutions than sequence-based compositions, which makes classical search methods not applicable for this kind of web services composition. Fig. 2.1 shows the size of the search space for both sequence and workflow-like compositions and two different services repositories with sizes 100 and 8,000. The x-axis represents the depth of the tree for workflow-like compositions and the corresponding search space size has been calculated using the context-free grammar defined in this paper (Fig. 2.2). The size of the search space has been calculated in a precise way, generating all the valid structures for each tree depth and calculating the number of different compositions using the number of services of the corresponding services repository. Even if the number of services used for workflow-like compositions is 80 times lower than that of a sequence-based approach (100 vs. 8,000), the size of the search space for a depth of five is larger for workflow-like compositions.

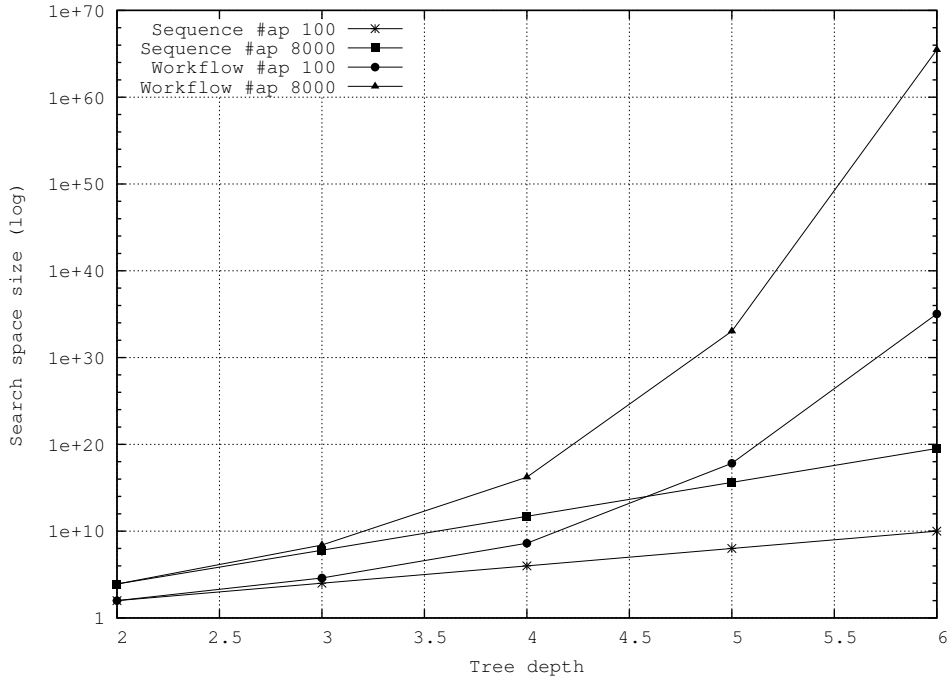

Figure 2.1: Search space size of sequence and workflow-like structures for different services repository sizes.

2.4 Related Work

Web services composition has attracted widespread attention in recent years. Although there are several proposals to classify the approaches that focus on this topic [6,36], in this paper we distinguish two kinds of algorithms depending on the complexity of the problem they solve:

1. Algorithms that solve the problem of generating *services sequences* whose execution leads to the desired result.

2. Algorithms whose aim is to obtain a workflow composed by a set of control structures that coordinate the service execution. Usual control structures in most workflow languages are sequences, parallel executions, synchronizations, selections and loops. The complexity of this kind of services compositions is higher than the sequence generation (Fig. 2.1) because, to achieve a solution, they must be taken into account the dependencies among control structures and how each structure deals with input/output data.

2.4.1 Services Sequence-based Composition

An extensive research in services composition has been focused on planning-based approaches in which the composition is modeled as a planning problem [45,72]. In these approaches there is an initial state defined by a set of both inputs and preconditions that the composite service must verify; a set of operators (or *services*) that are executed to obtain new and intermediate states; and a final state defined by a set of both outputs and postconditions that the solution must also verify. The composite service is therefore generated by a sequence of services whose ordered execution allows to achieve the requested outputs from the inputs.

Following this general model, different planners have been applied, such as graph analysis-based planners [50,126], where the GraphPlan [18] algorithm is adapted to find services compositions with optimal paths from inputs to outputs; logic-based planners [91], where the reasoning capabilities of a logic paradigm are used to obtain the services whose execution is compliant with the description of the state where they are applied to; hierarchical planners [60, 69, 110], where the hierarchical representation of composite services is considered to reduce the complexity in generating automatic sequences at different hierarchy levels; or planning as model checking [9, 17, 86], where the non-determinism and partial observability of services is managed for the generation of compositions. In these approaches, as for hierarchical planners, it is necessary to have an abstract representation of the workflow that

models the composite service (with abstract service descriptions). The planner has to select the concrete services that better fit to the predefined compositions.

The main drawback of these approaches is their low performance when *the search space is huge*, that is, when the number of services and the input/output interactions among them is high. In this case the number of operators (services) that could be applied to a given state (verifying partially its inputs and preconditions) is high and, therefore, the number of potential intermediate states is huge. In this situation, finding a solution is a hard problem that requires the use of optimal search techniques. To deal with this issue some strategies have been proposed:

- In [73] the planning algorithm is combined with regression search to minimize the *number of services* that could be applied to a given state. Thus, once a services sequence is obtained, an heuristic greedy search is applied in a backward sense to approximate the optimal sequence of services.
- In [93] a query index with semantic information about inputs/outputs concepts of the services is created in order to reduce the reasoning time needed to obtain a matching between the inputs and outputs of the services.

The other disadvantages of these approaches are that: (*i*) they have not been validated in large services repositories; and (ii) the generation of services sequences, usually, has not the optimal execution path, because parallel structures are not considered as part of the solution. However, as implicit loops are allowed in the algorithm, a solution to this issue would be to apply a pattern matching algorithm to discover control structures in the sequences [50].

2.4.2 Automatic Workflow Composition

Several approaches based on evolutionary algorithms have been proposed to obtain services compositions whose description is carried out through workflows [12, 26, 117, 127]. For example, [12] describes an algorithm for services composition in BPEL that follows a similar approach to the one presented in this paper. The main differences with our proposal are that: (*i*) it does not show a formal description of the grammar to compound services; (*ii*) attributes updating after crossover and mutation is not explicitly managed. Therefore, it is difficult to evaluate to which degree all the interactions among services are fulfilled to get a correct solution; (*iii*) minimum execution paths are not assured because this parameter is not included as part of the fitness function; and (*iv*) the algorithm has been validated in a private repository.

In [26], authors consider a workflow of tasks and a set of services that may execute each of those tasks. The proposal presents an evolutionary algorithm to associate a task with an optimal service, and considers a fitness function implemented as a multi-objective and distance-based algorithm that evaluates quality of service parameters. In this algorithm, therefore, it makes no sense to include the minimum execution path of the composite service as a criterion to select individuals, because the workflow is predefined. A similar approach has been presented in [117], where the fitness function is calculated as a formula with weights for the different quality of service parameters.

In [127] a particle swarm optimization algorithm is applied to optimize the selection of services that are part of the solution. In order to do that, the semantic similarity between the service characteristics is calculated, obtaining a set of measures (or distances) that define the relation between a service and the other services of the repository. When these measures are available, the algorithm obtains a services sequence with optimal distances among the services. This work has not been validated in a large repository; it used the Amazon services to demonstrate the viability of the algorithm.

Furthermore, in the bibliography many other approaches for composition of service workflows have been proposed, approximating the solution with different search strategies such as heuristic search [3] or graph analysis [134]. Common drawbacks of these proposals is that they cannot manage all the control structures as are defined in workflow languages as BPEL4WS and OWL-S, and the performance of the algorithm decreases as the number of services and interactions among them is huge.

With this state of the art, we can conclude that the main differences between other approaches and our proposal are:

- Current approaches focusing on automatic generation of workflows do not consider all the control structures of the workflow languages like OWL-S and BPEL4WS. Thus, planning algorithms only obtain services sequences and evolutionary or optimization techniques do not manage the complete set of workflow-like structures.

- Some approaches do not minimize the execution path needed to execute the composite service, that is, they do not maximize the use of parallel control structures to reduce execution times.

- Existing proposals have not been validated in several repositories with different features in order to demonstrate the generality of the algorithm.

All these drawbacks have been tackled by our genetic programming-based approach to web services composition, which is described in the next section.

2.5 Genetic programming for web services composition

Web services composition requires the combination of many atomic services using several control structures. This combination of elements can be modeled, in a natural way, with a tree that represents the solution to a web services composition. As not all the combinations of atomic services and control structures are valid from a syntactical point of view, restrictions in the syntactical structure of a solution (web services composition) can be described with a context-free grammar. Genetic programming is especially adequate for web services composition due to:

- Genetic programming can deal with solutions with very different structures as the individuals are usually represented by trees and, moreover, the trees can have different depths and number of nodes.

- A context-free grammar can be naturally included to generate new individuals, and to produce right structures for the individuals after crossover and mutation.

- Web services composition has a hierarchical structure, i.e., several atomic services generate a composite service, several composite services produce a more complex composition and so on, until the desired solution is found. Therefore, the subtrees of a tree represent simple compositions that contribute to the solution. Intermediate compositions can be interchanged between trees, in order to improve the performance of the new trees (solutions). This is exactly what is implemented with the crossover operator in genetic programming.

The first step in the design of an algorithm for web services composition requires the definition of the type of composite services that are going to be build. A compact definition of the valid structures of a tree (chromosome) for a web services composition can be described by a context-free grammar.

2.5.1 Context-free grammar

A context-free grammar is a quadruple (V, Σ, P, S), where V is a finite set of variables, Σ is a finite set of terminal symbols, P is a finite set of rules or productions, and S is an element of V called the start variable. The grammar that defines the valid structures for web services composition is described in Fig. 2.2. The first item enumerates the variables, then the terminal symbols, in third place the start variable is defined, and finally the rules for each variable are enumerated. When a variable has more than one rule, rules are separated by symbol "|".

- $V = \{$*initialProcess, process, compositeProcess*$\}$
- $\Sigma = \{$*atomicProcess, choice, sequence, split, splitJoin*$\}$
- $S =$ *initialProcess*
- Rules:
  - <*initialProcess*>::= <*compositeProcess*>| *atomicProcess*
  - <*process*>::= <*compositeProcess*><*process*>| *atomicProcess* <*process*>| <*compositeProcess*>| *atomicProcess*
  - <*compositeProcess*> ::= *choice* <*process*> <*process*> | *sequence* <*process*> <*process*> | *split* <*process*> <*process*> | *splitJoin* <*process*> <*process*>

Figure 2.2: Context-free grammar for web services composition.

The grammar has been defined to fulfill the syntax of the most common web services composition languages (OWL-S and BPEL4WS), and is completely independent of the services repository. <*initialProcess*> is the start variable of the grammar and generates an atomic or a composite process.

Variable <*process*> defines either composite processes or atomic processes. Two of the four rules of this variable are recursive and, therefore, a process can be composed of any number of atomic and composite processes. Finally, variable <*compositeProcess*> represents the combination of a control structure and two processes (of any type), i.e., a composition of at least two processes.

All the nodes of type variable, together with terminal symbol *atomicProcess* constitute the service nodes. They are characterized by the following attributes:

- Control structure: the node of type control structure ($\{$*choice, sequence, split, splitJoin*$\}$) of which the service node depends on. The control structure manages the interaction among the services that share that control.

- Available inputs: are those inputs available for a service. A subset of them are selected as inputs to the service. An input can be available in two ways. First, if the user introduces that input. In second place, if a service that belongs to the composition and has been executed before (in the composition flow), generates as output that service functionality.

- Necessary inputs: are the inputs that the node needs for running all the atomic processes in the subtree for which the node is the root node. These inputs or their subclasses have to be provided by the user or by other services of the composition.

- Obligatory inputs: in some situations, the outputs of several services have to be used as inputs to the current service. This means that at least one of those outputs has to be selected as input to the current service (a semantic matching among thme must exist). An example of this situation is the sequence of two services S_a and S_b. Let $O_a = \{o_1^a, \ldots, o_{n_a}^a\}$ be the set of outputs generated by service S_a, and $I_b^n = \{i_1^b, \ldots, i_{n_b}^b\}$ be the set of necessary inputs of S_b. Then, $O_a \cap I_b^n \neq \emptyset$. If this condition is not fulfilled, the composition of services S_a and S_b is not a sequence, and the structure is not valid. Therefore, the inputs of the service must contain a subset of the obligatory inputs. Following the example, the obligatory inputs of service S_b are the outputs of service S_a, i.e., $I_b^o = O_a$.

- Outputs: generated by the service. They can be directly generated by the service (if it is an atomic process) or by the subtree with the service as root node (composite process).

2.5.2 Attributes updating

The initialization of a tree (web services composition), or a modification of it due to crossover or mutation, requires the updating of all the attributes of each node. The initial step of the algorithm resets all the attributes of all the nodes in the tree, and then initializes the necessary inputs of the root node (<*initialProcess*>) to the set of inputs of the web services composition to be solved. Then, the tree is traversed in preorder, updating the attributes of each node. To traverse a tree in preorder, the following operations must be performed recursively at each node, starting with the root node: first, visit the root. Then, traverse the subtrees that have as root node the children of the root. Children are traversed in order, starting with the leftmost node and continuing to the right. Updating the attributes of each node is done in a different way depending on the type of attribute:

- Control structure (cs): this attribute is propagated in a top-down way. This means that a node inherits the attribute value from its parent. There is an exception to this rule. The node will set its control structure to its leftmost brother when that brother is a control structure.

- Available inputs (I^a): they are propagated in a top-down way. If an input is available for a node, it will also be available for all its children. When the control structure of the node is *sequence*, all the outputs of the brothers to the left of the node will also be added as available inputs.

- Necessary inputs (I^n): the propagation is done in a bottom-up way. This means that, if and only if the node is a leaf node, all its ancestor nodes will add as necessary inputs the necessary inputs of the node.

- Obligatory inputs (I^o): they are propagated in a top-down way. When the control structure of the node is *sequence* and the brother node immediately to the left is a service node, the obligatory inputs will be set with the following algorithm:

    1. Traverse in preorder the subtree that has as root node the brother node just to the left of the current node (the one for which the obligatory inputs are being calculated).
    2. Get the last node traversed in that process. It will be the rightmost node of the subtree.
    3. If both the last and current nodes depend on the same control structure (they have a reference to the same node of type *controlStructure*), then the outputs of the last node will be the obligatory inputs of the current node.
    4. Else, the outputs of the brother node immediately to the left will be the set of obligatory inputs of the current node.

- Outputs (O): the attribute is propagated in a bottom-up way (the outputs of a node will also be outputs of its parent), except when the leftmost child of the node is a *choice* control structure. Outputs for this situation are obtained as: $O = O_1 \cap \ldots \cap O_n$, i.e., the intersection of the outputs of all the children of the node.

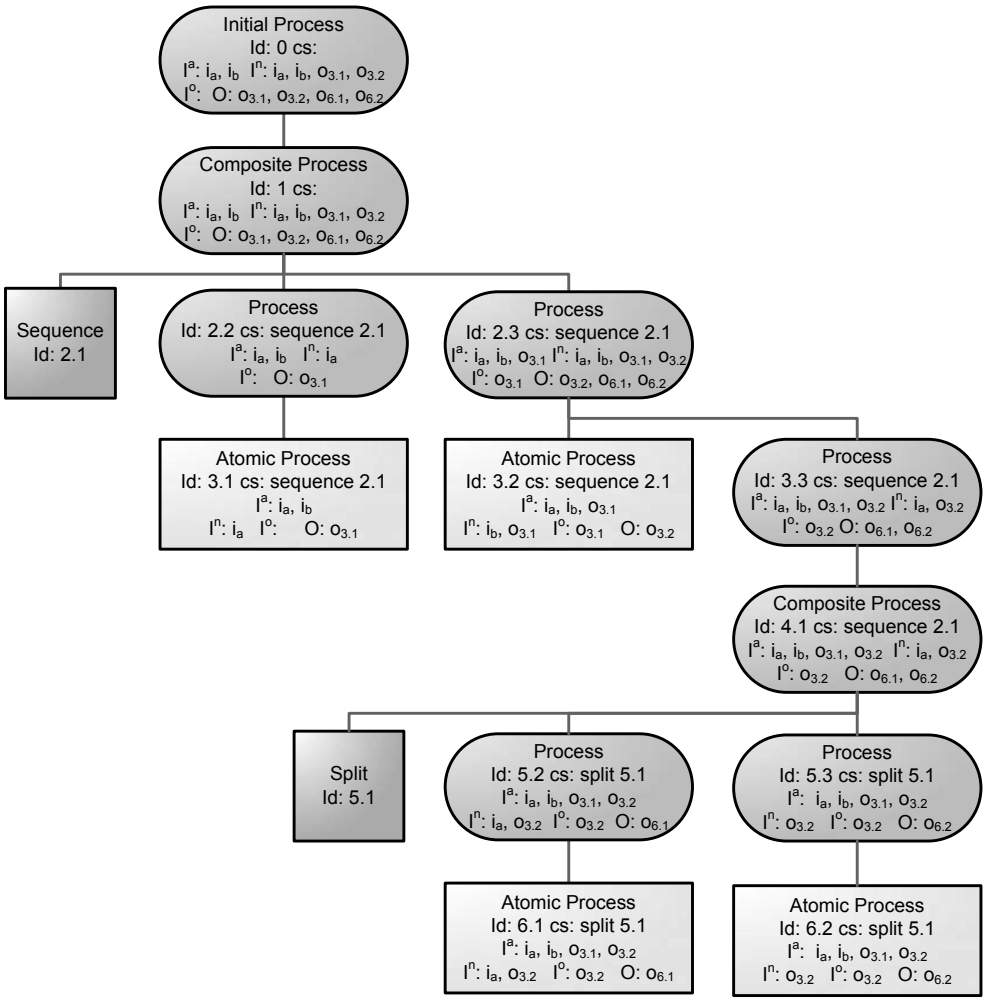

Figure 2.3: A chromosome representing the composition of several atomic processes.

An example

Fig. 2.3 shows a services composition. Terminal symbols (leaves of the tree) are represented by rectangles or squares, and variables are shown as flatted circles. Each node includes the values of the different attributes: the control structure governing the node (*cs*), the available

inputs (I^a), the necessary inputs (I^n), the obligatory inputs (I^o) and the outputs (O). In this example the initial available inputs are i_a and i_b, and the outputs required to solve the composition are $o_{6.1}$ and $o_{6.2}$.

Attributes updating starts from the root node, traversing the tree in preorder. When the first atomic process node (3.1) is reached, its available and obligatory inputs are set to its parent values, which were also taken from its ancestor (top-down updating)). This service uses i_a as input and generates $o_{3.1}$ as output. Therefore, the necessary inputs and the outputs will be set to these values and propagated to all the ancestors of the node.

Following the preorder traversal, node 2.3 is visited. As this node has a *sequence* control structure and has brother service nodes on the left, both the available and obligatory inputs require a different updating. The available inputs are those inherited from the parent (top-down updating) plus all the outputs generated by the brother nodes to the left, i.e., $o_{3.1}$ is added as an available input. On the other hand, the obligatory inputs are the outputs of the brother node ($o_{3.1}$). These attribute values are propagated down to node 3.2. This node is an atomic process that generates output $o_{3.2}$ using inputs i_b and $o_{3.1}$. Attributes output and necessary inputs are consequently updated and propagated to its ancestors.

The next traversed node is 3.3. Again, this node has a *sequence* control structure and has brother service nodes on the left. Therefore, the output of node 3.2 is added as available input to the node and, also, the obligatory inputs attribute is set to this value.

Both the available and obligatory inputs are propagated down. Thus, nodes 6.1 and 6.2 have to use $o_{3.2}$ as input. Both nodes propagate up the necessary inputs (i_a, $o_{3.2}$) and the outputs ($o_{6.1}$, $o_{6.2}$), and the updating process ends with the configuration shown in Fig. 2.3.

2.5.3 Genetic programming-based algorithm

Fig. 2.4 describes the genetic programming algorithm that has been used for web services composition. The first three steps of the algorithm correspond to an initialization. t represents the number of iterations, while *timesRun* will be used to detect situations in which the search gets stuck. The iterative part of the algorithm starts at step four. This part will be repeated until the maximum number of iterations is reached or the best possible solution is found. The main stages of the iterative part are the selection of the individuals, the crossover and mutation to generate new individuals, the post-processing, their evaluation, the replacement of the population, the local search, and the checking of stuck situations in the search process. All of them are described in detail in the next sections.

1. Initialize population
2. Evaluate population
3. $t = 1$, *timesRun* = *initialTimesRun*
4. While $t \leq$ *maxT* and $fitness_{best} <$ *maxFitness*
   a) Selection
   b) Crossover
   c) Mutation
   d) Post-processing
   e) Evaluate new individuals
   f) Replace population
   g) Run the local search
   h) $t = t + 1$
   i) If $bestInd(t) = bestInd(t-1)$, then *timesRun* = *timesRun* $- 1$
   j) If none of the individuals of the population have been created in the current iteration, then *timesRun* = *timesRun* $- 1$
   k) If *timesRun* < 0, then:
      i. Reinitialize population, keeping only the best individual.
      ii. Evaluate new individuals
      iii. *timesRun* = *initialTimesRun*
5. Final post-processing

Figure 2.4: Genetic programming algorithm for web services composition.

Initialization

The first step of the algorithm is the generation of the initial population. A new individual is generated applying randomly the rules of the grammar. If the depth of the tree reaches the maximum predefined value, then all the nodes of type service at that depth are transformed to *atomicProcess* nodes. Once the structure of the tree has been defined, the attributes of the nodes must be initialized using the algorithm defined in Sec. 2.5.2.

This attributes updating algorithm is run with one special characteristic. When an *atomicProcess* node is reached during the traversal of the tree, as no specific service has been assigned to it, one has to be selected from the repository. The selection is done randomly from the set of services that fulfill: $I_j^a \supseteq I_k$ and $I_j^o \cap I_k \neq \emptyset$. Thus, a service k can be selected if its inputs are a subset of the available inputs of the *atomicProcess* node j (I_j^a) and if at least one of the inputs

of k belongs to the set of obligatory inputs of j (I_j^o).

Evaluation

The calculation of the fitness of each individual of the population is done analyzing four criteria: generated outputs, used inputs, execution time of the composite service and number of nodes of type *atomicProcess*:

$$fitness = \omega_1 \cdot \left(\frac{\sum_i^{|O_{obj}|} \frac{1}{DO_i+1}}{|O_{obj}|} + \frac{|I^n_{root} \cap I_{obj}|}{|I_{obj}|} \right) + \omega_2 \cdot \frac{1}{runPath} + \omega_3 \cdot \frac{1}{\#atomicProcess} \quad (2.1)$$

where O_{obj} are the outputs that are required to solve the composition, DO_i is the distance of the individual to the i-th required output, I^n_{root} are the necessary inputs of the root node (this node is the result of the composition of the services), I_{obj} are the inputs provided to solve the composition, *runPath* is the execution time of the composite service, *#atomicProcess* is the number of atomic processes in the tree, and ω_k are values that weight the importance of each criterion.

The first and second criteria indicate the degree to which a valid solution has been found. The first one is the number of outputs (or subclasses of them), of those that were required, that have been generated by the composition. The second criterion is the number of inputs (or superclasses of them), of those provided by the user, that have been used.

In order to guide the composition, the use of a crisp criterion for the outputs is not adequate, and the concept of distances to outputs (DO_i) must be introduced. For example, if the expected result of a composition is a sequence of ten services, and the desired output is provided by the last service, a composition of the first nine services will not generate the desired output and, therefore, with a crisp criterion, this part of the fitness function would be evaluated as 0. However, if the fitness function measures the distance between the composite service (of nine atomic processes) and the desired output, it will reflect that the composite process is close to find a valid solution (only a new atomic service needs to be added). The distance of an individual to the i-th desired output (DO_i) is calculated in the following way:

$$DO_i = \min_j DO_{ij} \quad (2.2)$$

where DO_{ij} is the distance of the j-th atomic service of the individual to the i-th output. Thus, the distance of the individual to the output is the minimum of the distances of its atomic

services to that output. Also,

$$DO_{ij} = \min_{k} DO(S_j, S_k) : o_i \in O_k \tag{2.3}$$

where S_j and S_k are services, O_k is the set of outputs generated by service S_k, and o_i is the i-th output. DO is the distance between two atomic services, and is defined as the minimum number of atomic services that need to be composed in sequence, starting with S_j, in order to generate an output of S_k. For example, if $O_j \cap O_k \neq \emptyset$, then $DO(S_j, S_k) = 0$.

The third criterion is the execution time of the composite service. This time depends on the execution time of each atomic service but, also, on the control structures in the following way:

- *sequence*: the execution time is the sum of the times of all the services in the sequence.
- *split* and *splitJoin*: the execution time is equal to the time of the slowest service belonging to this control structure, as all the services are executed in parallel.
- *choice*: in this control structure, only one service of the composition is executed. As the selected service is only known at run time, the worst time of all the services in the *choice* composition has to be selected. Therefore, the execution time is calculated in the same way as for the *split* control structure.

Finally, the last criterion is related with the complexity of the composite service. The higher the number of atomic processes in the composition, the higher the complexity.

Selection

The selection mechanism that has been used is the binary tournament selection. In a k-tournament selection, k individuals are randomly picked from the population with replacement, and the best of them is selected. In this case, $k = 2$ (binary tournament selection).

Crossover

The crossover operator replaces a subtree of an individual with a subtree of other individual. The process is as follows:

- Select randomly a node of type service in the first individual.

- Generate the set of candidate nodes in the second individual. These nodes must have the following characteristics:
  - They must be of type service.
  - $(I_2^n - O_2) \cap I_1^o \neq \emptyset$. $I_2^n - O_2$ represents all the inputs that are used by the subtree of the second individual and that have not been generated inside that subtree. This set of inputs must contain at least one of the obligatory inputs (or their subclasses) of the subtree that is going to be replaced in the first individual.
  - $I_2^n - O_2 \subseteq I_1^a$. Also, the set of inputs used in the subtree of the second individual must be a subset of the available inputs (or their subclasses) for the subtree of the first individual.
- Select randomly a node of the candidate nodes set, and replace the subtree of the first individual with the selected subtree of the second individual.
- Execute the attributes updating algorithm. During the execution of the algorithm, if a leaf node of type *atomicProcess* is reached, two conditions must be checked: $I_j^a \supseteq I_k$ and $I_j^o \cap I_k \neq \emptyset$, i.e., the inputs of the process (I_k) must be a subset of the available inputs (or subclasses of them) of the node and, also, they must contain at least one of the obligatory inputs (or their subclasses) of the node. If the conditions are not fulfilled, a new atomic process must be selected using the same procedure as in the initialization stage (Sec. 2.5.3).

Mutation

The mutation operator modifies a subtree of the individual. First, a node must be randomly selected. If the node is of type variable, then the subtree that has as root the selected node is eliminated. The new subtree is generated applying the rules of the grammar randomly for that variable in the same way as in the initialization stage (Sec. 2.5.3). On the other hand, if the node is of type terminal, there are two cases:

- If the node is a control structure, a new one is randomly selected.
- If the node is an atomic process, there are two posibilities:
  - A new process is randomly selected from the repository using the same conditions defined in the initialization stage (Sec. 2.5.3)

- The node is substituted with a process node and a subtree is generated applying the rules of the grammar in the same way as for nodes of type variable.

In all the cases, the attributes updating algorithm must be run. Also, the validity of the atomic processes must be checked and, if necessary, a new selection of the atomic processes is done.

Replacement

The selection mechanism is a population-based selection approach, i.e., parents and their corresponding offspring are combined generating a population with a size $2N$ (being N the size of the initial population), and the best N individuals are selected for the next population.

Post-processing

The size and complexity of the trees representing the individuals has to be managed in order to improve the search, reduce the time per iteration and, also, to simplify the final composite service. The post-processing stage consists of four steps that are executed at the end of the algorithm. Moreover, two of these steps are also executed at the end of each iteration. The steps must be executed in the following order:

1. *Eliminate useless atomic services*: an atomic service is useless if none of their outputs neither contribute to the objective outputs nor are inputs to other services. Elimination of this kind of services is recursive, i.e., it is repeated while in the previous step a service was eliminated. New useless services can appear due to the elimination of a useless service. This procedure is executed at the end of the algorithm.

2. *Eliminate useless control structures*: a control structure is useless when only one atomic service depends on that control structure. A control structure is used to compose services and, therefore, a minimum of two services are needed. Useless control structures have to be eliminated, and the atomic process belonging to it is assigned to the control structure of its closer ancestor. This step needs to be done at the end of each iteration.

3. *Eliminate consecutive and equal control structures*: when a node and its parent have the same type of control structure and it is an *split* or an *splitJoin*, both control structures can be merged. This step is executed only at the end of the algorithm.

4. *Tree flatten*: the depth of the trees is limited in order to prevent an infinite growth of the individuals. The proposed context-free grammar is unambiguous, and this means that an individual can be only represented by a tree. However, due to crossover, mutation and, also, due to the recursive rules in the grammar, the number of nodes and the depth of the trees can grow at a high rate. In large trees, usually some of the internal nodes are useless, i.e., they could be deleted without modifying the fenotype of the individual (the composition remains equal), although the genotype is changed. This process generates smaller individuals, which improves the search for better compositions. Tree flatten transforms a tree in its equivalent with the minimum depth, and it is done in each iteration. To keep the same fenotype, it is necessary to respect the precedence among the different control structures in the individual. The process is started in the leaf nodes and ends when the root node is reached. Each node in the tree is gone up to the depth of its control structure node.

Local search

The objective of the local search is to improve some of the individuals of the population implementing a search process with a low degree of exploration and a high degree of exploitation, i.e., a very exhaustive search in the neighborhood of the individual. In this case, the local search has been applied to only one individual of the current population: the best individual. If the local search was already run for that individual in previous iterations, then a new individual is randomly selected to execute the local search.

The local search algorithm is described in Fig. 2.5. It is a greedy algorithm (proceeds by changing the current assignment by always trying to increase the fitness) called steepest ascent hill climbing [104]. This algorithm fulfills two requirements that are necessary for its adequate cooperation with the genetic programming algorithm for web services composition: it makes a complete search in the neighborhood of the solution until it finds the local maximum, and it is very fast. A loop (lines 2-14) is run until the local search fails to improve the best solution in one iteration. The best solution of the iteration (Ω') is initially set to the best solution (Ω). For all the neighbors of the the best solution, the best one is picked if it improves the best solution of the iteration (lines 5-9). Finally (lines 10-13), if the best solution of the iteration improves the best solution, the best solution is updated and the local search continues. Else, the best solution is returned and it will replace the original solution in the population if it is better.

The local search requires the generation of the neighbors of an individual (line 5). The neighborhood can be obtained substituting each atomic process of the individual with another atomic process from the repository (fulfilling some conditions that will be detailed later). As there can be several candidates for each replacement and there are also several *atomicProcess* nodes in the individual, making all the combinations can generate a huge number of neighbors. Thus, in order to speed up local search, a reduced number of neighbors will be generated as follows:

```
1: Obtain the initial solution, Ω.
2: repeat
3:     Ω' = Ω
4:     continue = false
5:     for all neighbors Ω'' of Ω do
6:         if fitness(Ω'') > fitness(Ω') then
7:             Ω' = Ω''
8:     if fitness(Ω') > fitness(Ω) then
9:         Ω = Ω'
10:        continue = true
11: until continue
12: Return Ω
```

Figure 2.5: Steepest ascent hill climbing algorithm [104].

1. Select randomly the number of *atomicProcess* nodes of the individual that will be modified ($\#ap_{LS}$).

2. Pick randomly those *atomicProcess* nodes that will be modified.

3. For each node of type *atomicProcess* (ap_j) that has been picked, look for all the processes in the repository that fulfill the follwing conditions:

   a) $I_j^a \supseteq I_{jk}^n$

   b) $I_j^o \cap I_{jk}^n \neq \emptyset$

   c) $O_{jk} \supseteq O_j^n$

   where I_{jk}^n are the necessary inputs of process ap_{jk}, $k = 1, \ldots, \alpha_j$. ap_{jk} is the k-th atomic process of the repository that fulfills the conditions for node j (which corresponds to atomic process ap_j), O_{jk} are the outputs of ap_{jk}, and O_j^n are the outputs that were generated by atomic process ap_j and that were used as inputs by other atomic processes of the individual.

4. Calculate for each ap_{jk} the probability to be selected: $p_{jk} = \eta\, fitness_{jk}$, where $fitness_{jk}$ is the fitness of the individual after the replacement of the atomic process of node j by ap_{jk}, and η is a normalization factor.

5. For each considered ap_j, pick randomly one of the ap_{jk} using the calculated probabilities (p_{jk}).

6. The neighborhood is composed of all the individuals obtained after replacing or keeping the corresponding nodes (ap_j). The size of this neighborhood is $2^{\#ap_{LS}} - 1$.

Reinitialization

The last steps of each iteration (Fig. 2.4) update the value of *timesRun*, decreasing it when the best individual has not improved and also when no individuals of the current iteration have survived the replacement process. If *timesRun* takes a value below 0, the population is reinitialized in the same way as in the initialization stage, but keeping the best individual.

2.6 Results

```
1. Obtain the time interval and the diagnostic process for a hospital:
     • Inputs:  _HOSPITAL
     • Outputs:  _TIMEINTERVAL, _DIAGNOSTICPROCESS
     • Solution:  HOSPITAL_DIAGNOSTICPROCESSTIMEINTERVAL_SERVICE
     • List of atomic processes:
         – HOSPITAL_DIAGNOSTICPROCESSTIMEINTERVAL_SERVICE:
             * Inputs:  _HOSPITAL
             * Outputs:  _TIMEINTERVAL, _DIAGNOTICPROCESS
2. Confirm if, given a town, country and a price, it is possible to
   buy coffee and whiskey:
     • Inputs:  _COUNTRY, _TOWN, _RECOMMENDEDPRICE
     • Outputs:  _COFFEE, _WHISKEY
     • Solution:  sequence(TOWNCOUNTRY_HOTEL_SERVICE,
       HOTELRECOMMENDEDPRICE_COFFEEWHISKEY_SERVICE)
     • List of atomic processes:
         – TOWNCOUNTRY_HOTEL_SERVICE:
             * Inputs:  _COUNTRY, _TOWN
```

```
            * Outputs:  _HOTEL
        - HOTELRECOMMENDEDPRICE_COFFEEWHISKEY_SERVICE:
            * Inputs:  _RECOMMENDEDPRICE, _HOTEL
            * Outputs:  _COFFEE , _WHISKEY
3. Obtain the maximum price of a book given the academic item number
   of the author:
   • Inputs:  _ACADEMIC-ITEM-NUMBER
   • Outputs:  _MAXPRICE, _BOOK
   • Solution:  sequence(ACADEMIC-ITEM-NUMBER_BOOKAUTHOR_SERVICE,
     AUTHOR_BOOKMAXPRICE_SERVICE)
   • List of atomic processes:
     - ACADEMIC-ITEM-NUMBER_BOOKAUTHOR_SERVICE:
       * Inputs:  _ACADEMIC-ITEM-NUMBER
       * Outputs:  _AUTHOR , _BOOK
     - AUTHOR_BOOKMAXPRICE_SERVICE:
       * Inputs:  _AUTHOR
       * Outputs:  _MAXPRICE , _BOOK
4. Get the maximum price of a book, its type and the recommended price
   in dollars using the academic item number of the author:
   • Inputs:  _ACADEMIC-ITEM-NUMBER
   • Outputs:  _MAXPRICE , _BOOK-TYPE , _RECOMMENDEDPRICEINDOLLAR
   • Solution:  sequence(ACADEMIC-ITEM-NUMBER_BOOKAUTHOR_SERVICE,
     split(AUTHOR_BOOKMAXPRICE_SERVICE,
     BOOK_RECOMMENDEDPRICEINDOLLAR_SERVICE,
     BOOK_AUTHORBOOK-TYPE_SERVICE))
   • List of atomic processes:
     - ACADEMIC-ITEM-NUMBER_BOOKAUTHOR_SERVICE:
       * Inputs:  _ACADEMIC-ITEM-NUMBER
       * Outputs:  _AUTHOR , _BOOK
     - AUTHOR_BOOKMAXPRICE_SERVICE:
       * Inputs:  _AUTHOR
       * Outputs:  _MAXPRICE , _BOOK
     - BOOK_RECOMMENDEDPRICEINDOLLAR_SERVICE:
       * Inputs:  _BOOK
       * Outputs:  _RECOMMENDEDPRICEINDOLLAR
     - BOOK_AUTHORBOOK-TYPE_SERVICE:
       * Inputs:  _BOOK
       * Outputs:  _BOOK-TYPE
5. Get the weather, map and hotel given the city:
   • Inputs:  _CITY, _DURATION, _COUNTRY
   • Outputs:  _WHEATHERSEASON, _MAP, _HOTEL
```

- Solution: split(CITYCITY_MAP_SERVICE, CITY_WHEATHERSEASON_SERVICE, DURATIONCOUNTRYCITY_HOTEL_SERVICE)
- List of atomic processes:
  - CITYCITY_MAP_SERVICE:
    * Inputs: _CITY
    * Outputs: _MAP
  - CITY_WHEATHERSEASON_SERVICE:
    * Inputs: _CITY
    * Outputs: _WHEATHERSEASON
  - DURATIONCOUNTRYCITY_HOTEL_SERVICE:
    * Inputs: _CITY, _DURATION, _COUNTRY
    * Outputs: _HOTEL

Figure 2.6: Description of the web services compositions used for testing on repository *OWL-S TC V2.2*.

1. Web Service Challenge testset 1:
   - Inputs: con1233457844, con1849951292, con864995873
   - Outputs: con1220759822, con2119691623
   - Solution: sequence(splitJoin(serv75024910, serv1599256986, serv1668689219), serv976005395, serv283321609, splitJoin(serv1738121452, serv1114869861), serv1876985918, split(serv1184302094, serv491618308))
   - List of atomic processes:
     - serv75024910
       * Inputs: con1653328292, con1849951292, con241744282
       * Outputs: con1211952995, con1482103504
     - serv1599256986
       * Inputs: con1653328292, con1849951292, con241744282
       * Outputs: con100012944, con1810216552, con406825148
     - serv1668689219
       * Inputs: con1653328292, con1849951292, con241744282
       * Outputs: con1257011377, con95711533
     - serv976005395
       * Inputs: con1348154594, con424848942, con588701442, con848610623
       * Outputs: con1189013645, con134421950, con1399563071, con30170533, con51881517, con633555781
     - serv283321609
       * Inputs: con10304228, con1189013645, con30170533,

```
                con53520061
             * Outputs:  con1289781877, con1489681927, con149168694,
                con351525476, con730842958, con912923257
          - serv1738121452
             * Inputs:  con1489681927, con149168694, con1631823443,
                con666530324, con912923257
             * Outputs:  con1804686775, con1910780741, con556545125
          - serv1114869861
             * Inputs:  con1489681927, con149168694, con1631823443,
                con666530324, con912923257
             * Outputs:  con164119443, con189107477, con582761525
          - serv1876985918
             * Inputs:  con2129932951, con582761525, con764841824
             * Outputs:  con1498488754, con1869203452, con801503557,
                con851887673
          - serv1184302094
             * Inputs:  con2049645207, con323056349, con761564774
             * Outputs:  con1357575604, con365862042
          - serv491618308
             * Inputs:  con2049645207, con323056349, con761564774
             * Outputs:  con1335046394, con1772940636, con427511809
2. Web Service Challenge testset 2:
    • Inputs:  con1498435960, con189054683, con608925131,
      con1518098260
    • Outputs:  con357002459
    • Solution:  sequence(splitJoin(sequence(serv1189164894,
      serv496481108), serv1258597127, serv2020713184),
      serv1328029360)
    • List of atomic processes:
      - serv1189164894
         * Inputs:  con1233815228, con1498435960, con1518098260,
           con189054683
         * Outputs:  con2040171441, con2050616774, con2085025818,
           con915123190
      - serv496481108
         * Inputs:  con2050616774, con699658208, con915123190
         * Outputs:  con115731217, con1545953204, con276510710,
           con29503594, con395302698
      - serv1258597127
         * Inputs:  con1233815228, con1498435960, con1518098260,
           con189054683
```

```
            * Outputs:  con1531820643, con1609445520, con965917446
        - serv2020713184
            * Inputs:  con1233815228, con1498435960, con1518098260,
              con189054683
            * Outputs:  con1027771256, con140513116, con2027267284,
              con2085025818, con674466131, con699658208,
              con763764707, con781788463, con794896663
        - serv1328029360
            * Inputs:  con1038626729, con146453033, con395302698,
              con794896663, con798787896, con813330597, con918400240
            * Outputs:  con368472115, con669754580, con841389546
```

Figure 2.7: Description of the web services compositions 1 and 2 used for testing on repositories from *WSC 2008*.

```
1. Web Service Challenge testset 5:
    • Inputs:  con428391640, con2100909192
    • Outputs:  con1092196197, con1374634550, con2055848680
    • Solution:  sequence(serv247333572, splitJoin(serv316765805,
      serv386198038, serv1840997881), serv1217746328,
      splitJoin(serv525062504, sequence(serv2049294580,
      serv663926970), serv40675417, serv802791436),
      splitJoin(serv110107650, serv872223669, serv248972116,
      serv1703771959), splitJoin(sequence(serv1011088135,
      serv318404349), serv1080520368), split(serv387836582,
      sequence(serv1842636425, serv457268815)))
    • List of atomic processes:
        - serv247333572
            * Inputs:  con1368696763, con2100909192,
            * Outputs:  con1060243885, con1837312783, con1899780776,
              con28797675,
        - serv316765805
            * Inputs:  con1822359942, con384151484, con98229908,
            * Outputs:  con1196037436, con1275915002, con2140027657,
              con507448888, con6676551, con81844658,
        - serv386198038
            * Inputs:  con1822359942, con384151484, con98229908,
            * Outputs:  con1681242749, con17328019, con2082065080,
              con960704019,
        - serv1840997881
```

```
    * Inputs:  con1822359942, con384151484, con98229908,
    * Outputs:  con1082569052, con220709124, con369404740,
- serv1217746328
    * Inputs:  con1196037436, con1606076734, con1749856794,
      con359573590, con369404740, con418968538,
    * Outputs:  con1516982201, con1897732092, con361212134,
      con671505431, con82458841, con834129565,
- serv525062504
    * Inputs:  con131614591, con1602799684, con361212134,
      con844574898,
    * Outputs:  con215587433, con254912033,
- serv2049294580
    * Inputs:  con131614591, con1602799684, con361212134,
      con844574898,
    * Outputs:  con1121483512, con1464139261, con1548114195,
      con1944839158, con486968400,
- serv663926970
    * Inputs:  con1133159303, con1529884266, con2022463997,
    * Outputs:  con1269975085, con1310117911, con417330032,
      con524244316, con578519665,
- serv40675417
    * Inputs:  con131614591, con1602799684, con361212134,
      con844574898,
    * Outputs:  con1137664719, con1625739034, con1963069049,
      con459113456,
- serv802791436
    * Inputs:  con131614591, con1602799684, con361212134,
      con844574898,
    * Outputs:  con1116567918, con1412526779, con32688908,
      con414052982, con540015383,
- serv110107650
    * Inputs:  con1302131402, con1582113061, con1739819509,
      con1834035733, con1963069049, con27159169,
      con459113456,
    * Outputs:  con1120051103, con2077355659,
- serv872223669
    * Inputs:  con1302131402, con1582113061, con1739819509,
      con1834035733, con1963069049, con27159169,
      con459113456,
    * Outputs:  con308165151, con374114199, con427981500,
- serv248972116
```

```
    * Inputs:  con1302131402, con1582113061, con1739819509,
      con1834035733, con1963069049, con27159169,
      con459113456,
    * Outputs:  con1091786019, con1224710568, con1967574465,
      con292598089,
- serv1703771959
    * Inputs:  con1302131402, con1582113061, con1739819509,
      con1834035733, con1963069049, con27159169,
      con459113456,
    * Outputs:  con1210374002, con1395731351, con1928864048,
      con85939934,
- serv1011088135
    * Inputs:  con1070281170, con1818468709, con1882167160,
      con1967574465, con241599790, con427981500, con84711568,
    * Outputs:  con1620003160, con1753748027, con2101933515,
      con379850073, con76108784,
- serv318404349
    * Inputs:  con1141966130, con1753748027, con2101933515,
      con570327021,
    * Outputs:  con1759687944, con841297848,
- serv1080520368
    * Inputs:  con1070281170, con1818468709, con1882167160,
      con1967574465, con241599790, con427981500, con84711568,
    * Outputs:  con1119844968, con1264035168, con1613245017,
      con589171133,
- serv387836582
    * Inputs:  con1119844968, con1613245017, con1759687944,
      con658603366,
    * Outputs:  con1324864617, con1374634550, con1876431248,
      con240985607, con374934517, con424090267, con529978098,
      con793166421, con832491021,
- serv1842636425
    * Inputs:  con1119844968, con1613245017, con1759687944,
      con658603366,
    * Outputs:  con124240173, con1286564378, con1687592844,
      con495775189,
- serv457268815
    * Inputs:  con495775189, con952511413, con998186070,
    * Outputs:  con1740843832, con396235323, con465871599,
      con64435085, con885742047,
```

Figure 2.8: Description of the web service composition 5 used for testing on repositories from *WSC 2008*.

Table 2.1: Average results ($\bar{x} \pm \sigma$) for the test examples

Example	Search time (ms)	$\frac{\lvert I^n_{root} \cap I_{obj} \rvert}{\lvert I_{obj} \rvert}$	$\frac{\sum_i^{\lvert O_{obj} \rvert} \frac{1}{DO_i+1}}{\lvert O_{obj} \rvert}$	*fitness*	*runPath*	*#atomicProcess*
OWL-S TC V2.2–1	749.00 ± 364.10	1.00 ± 0.00	1.00 ± 0.00	1.0000 ± 0.0000	1.00 ± 0.00	1.00 ± 0.00
OWL-S TC V2.2–2	484.50 ± 139.20	1.00 ± 0.00	1.00 ± 0.00	0.9750 ± 0.0000	2.00 ± 0.00	2.00 ± 0.00
OWL-S TC V2.2–3	473.60 ± 76.19	1.00 ± 0.00	1.00 ± 0.00	0.9750 ± 0.0000	2.00 ± 0.00	2.00 ± 0.00
OWL-S TC V2.2–4	3010.20 ± 422.91	1.00 ± 0.00	1.00 ± 0.00	0.9296 ± 0.0042	2.20 ± 0.40	5.70 ± 1.19
OWL-S TC V2.2–5	1098.30 ± 240.72	1.00 ± 0.00	1.00 ± 0.00	0.9654 ± 0.0019	1.00 ± 0.00	3.30 ± 0.46
WSC 2008–1	6919.70 ± 1612.99	1.00 ± 0.00	1.00 ± 0.00	0.9112 ± 0.0012	6.00 ± 1.26	15.8 ± 5.71
WSC 2008–2	11137.30 ± 3106.75	1.00 ± 0.00	1.00 ± 0.00	0.9233 ± 0.0023	3.50 ± 0.67	6.00 ± 0.89
WSC 2008–5	95390.20 ± 43521.30	1.00 ± 0.00	1.00 ± 0.00	0.9069 ± 0.0011	9.20 ± 2.96	49.90 ± 16.84

The validation of the genetic programming algorithm for web services composition has been done with a set of experiments with different degrees of complexity. Four different repositories have been used for test:

1. *OWL-S TC V2.2*, with 1,000 services described with the OWL-S profile[5].

2. Web Service Challenge 2008 (*WSC 2008*) repository 1, with 158 services represented in WSDL and whose inputs and outputs are semantically described[6].

3. Web Service Challenge 2008 (*WSC 2008*) repository 2, with 558 services represented in WSDL and whose inputs and outputs are semantically described.

4. Web Service Challenge 2008 (*WSC 2008*) repository 5, with 1,090 services represented in WSDL and whose inputs and outputs are semantically described.

The services compositions that have been tested are shown in Figs. 2.6-2.8. For each example, a short description of the task that the composite service solves is given. Also, the available inputs (those provided by the user) and the desired outputs are enumerated. Then, the solution to the requested service is indicated: it is a combination of control structures and atomic processes. In most of the cases there are a few possible best solutions, but only one has been indicated in Figs. 2.6-2.8. Finally, each of the atomic processes that are part of the solution are described. It should be noticed that, as the inputs and outputs of the repositories *WSC 2008* are semantically described, the names of the inputs and outputs of the atomic processes (Figs. 2.7 and 2.8) do not match up. For example, the output of a service in a

[5] http://projects.semwebcentral.org/frs/download.php/386/owls-tc2_2_rev_2.zip
[6] http://cec2008.cs.georgetown.edu/wsc08/downloads/ChallengeResults.rar

sequence could not be an input to the next service. This is because the input of the next service is a superclass of the previous output (semantics has to be taken into account).

Table 2.1 shows the results for all the test examples described in Figs. 2.6-2.8. Each row in the table represents the results of the evolutionary algorithm for a test example. As evolutionary algorithms are nondeterministic, the result of one run over an example is not meaningful. Thus, for each of them 10 runs were executed. The columns represent the time to obtain the best solution found by the algorithm, the percentage of provided inputs that have been used by the atomic processes, the degree of fulfillment of the required outputs, the fitness value, the execution time (*runPath*) of the composite service (the execution time of each atomic service has been established to 1) and the number of atomic processes of the tree. For each of these columns, two values are represented: $\overline{\chi}$ is the arithmetic mean over 10 runs, and σ is the standard deviation over the 10 runs, which reflects the robustness of the probabilistic algorithm to obtain similar results regardless the followed pseudo-random sequence.

The values that have been used for the parameters of the evolutionary algorithm are: $maxT = 100$, $initialTimesRun = 20$, population size $= 200$, crossover probability $= 0.9$, mutation probability $= 0.03$ (per gene), maximum depth of the tree $= 9$, $\omega_1 = 0.45$, $\omega_2 = 0.05$, $\omega_3 = 0.05$, percentage of the individuals to apply local search $= 0.5\%$.

The first thing that must be noticed is that the fitness is, in nearly all the cases, under 1, as the execution time of the composite service and the number of atomic processes in the tree are greater than one (Eq. 2.1). The performance of the algorithm is good, as in all the tests an acceptable solution has been found for all the runs. This means that $I^n_{root} \cap I_{obj} = I_{obj}$ and $O_{root} \cap O_{obj} = O_{obj}$. Also, the search times[7] that have been obtained are quite low, which is specially important for web services composition, as users require a fast answer to their query. Going into the details for each test:

- *OWL-S TC V2.2*–1: this test is very simple, as it is just an atomic process and not a services composition. However, it has been included to verify that also under simple conditions the algorithm works properly (the best fitness was always reached). The number of atomic processes is always the right one, while the depth is always the minimum possible one.

[7] These times have been obtained with an Intel Xeon(R) Quadcore E5320 1.86GHz processor with 8GB of RAM, and the algorithm was implemented in Java and run on Linux.

- *OWL-S TC V2.2–2*: in this example all the executions reached the best possible services composition (two atomic processes connected in a *sequence*).
- *OWL-S TC V2.2–3*: this example is similar to the previous one (a *sequence* of two services). The best values for all the objectives (inputs, outputs, execution time, and number of atomic processes) have been reached in all the runs.
- *OWL-S TC V2.2–4*: this composition requires the use of two nested control structures: a *sequence* and an *split*. This solution cannot be constructed with sequence-based compositions. The execution time of the composite process was most of the times the lower one (it was over only two times). The number of atomic processes has a higher variability, indicating that correct compositions have been obtained in many different ways. A valid composition was found in all the runs.
- *OWL-S TC V2.2–5*: this composition is an *split* of atomic processes. In all the runs, a valid composition was obtained. Moreover, the execution time was always the lower one (1). The number of atomic processes was also, most of the times, the minimum.
- *WSC 2008–1*: this composition is very complex, as it requires the *sequence* of 6 processes, three of them composite processes. These composite processes are constructed with *split* and *splitJoin* control structures (this is one of the possible solutions to this composition). In all the runs, a valid solution was found. Results show a very low variability in the execution time of the composite process. On average, the execution time (6) is the same of the solution shown in Fig. 2.7. The number of atomic processes is higher than expected (10), as other valid solutions have been found with more than 10 atomic services.
- *WSC 2008–2*: this is also a complex composition, with three nested control structures: a *sequence*, an *splitJoin*, and a *sequence*. The first *sequence* controls two services. The first of them is a composite service of type *splitJoin* of three services. Again, the first of them is a composite process of type *sequence* over two atomic processes. The composition algorithm was able to find a valid solution in all the runs. Moreover, the execution time of the composite process was very close to the minimum one (3), and also the number of atomic processes was close to the minimum (5).
- *WSC 2008–5*: this is the most complex composition of all the tests and, also, the repository is the largest one (1,090 services). One of the solutions to this composition (the

one described in Fig. 2.8) uses three different control structures (*sequence*, *split*, and *splitJoin*), some of them nested three times. There are a total of 9 control structures and 20 atomic processes. The solution is a *sequence* of seven processes. Five of them are composite processes of type *split* and *splitJoin*. Moreover, some of these processes have other composite processes nested. For example, the second *splitJoin* has four processes, and the second one is a *sequence* of two atomic processes. Of course, this composition cannot be obtained with sequence-based compositions. Although the complexity of the solution, the proposed composition algorithm was always able to find a valid solution. Moreover, the execution time was very low. On the other hand, the number of atomic processes was, on average, over the minimum one.

In summary, the performance of the algorithm is very good. The tests have been selected to cover different types of compositions, using several control structures. Moreover, the complexity of the tests is really high (three of them come from the *WSC 2008*), with up to nine control structures in a composition, control structures nested up to three times, and more than twenty atomic services in some of the obtained solutions. Although these complex tests, the composition algorithm was always able to find a valid solution: in the 80 runs the result were always valid. Also, the execution time of the obtained solutions (*runPath*) was the lowest (or close to the minimum), which reflects the ability of the algorithm to exploit the different control structures that it can manage. Finally, the number of atomic processes of the solutions was, in most of the tests, close to the minimum.

2.7 Conclusions

A genetic programming algorithm for web services composition has been presented. The algorithm is able to compound services using different control structures, generates compositions following a context-free grammar, and manages explicitly the attributes updating. A full validation has been done for eight different composition problems coming from four different repositories (three of them from the Web Service Challenge 2008) with 158, 558, 1,000, and 1,090 services, showing a very good performance. In all the tests and runs a valid solution was found, indicating that the algorithm is robust and reliable for different repositories. Moreover, the execution times of the obtained composite processes were also low, showing the ability of the algorithm to exploit the available control structures. Also, the search times of the evolutionary algorithm are quite low, allowing to use our proposal on-line.

Chapter 3

An optimal and complete algorithm for automatic web service composition

One of the main advantages of control-centric vs. data-centric approaches is that control-centric approaches are more expressive since they can encode a wide variety of composition patterns using different control structures such as sequences, splits, choices and loops, among many other control structures, that cannot be expressed in a data-centric approach. However, this expressivity also works against obtaining optimal compositions at run time due to the vast search space that these algorithms are required to explore, as shown in Chapter 2. Thus, in this work we address the composition problem from a data-centric perspective that is better suited for the fast composition of information-providing services. For this purpose, we develop a graph-based algorithm that firstly analyzes the semantic information of the services to generate a *service dependency graph* that contains all the relevant services for the composition request together with their valid input-output matches. Once the graph is generated, we optimize its size by detecting services that are equivalent or dominated in terms of their functional interface. Then, an A*-based algorithm is used to extract the optimal composition from the graph, minimizing the total number of services and the runpath (or length) of the solution. In order to further improve the scalability, we enhance the search process using an admissible state pruning optimization that detects redundant states during the search, i.e., combinations of services that are functionally equivalent or dominated. We also provide a comprehensive

validation of the algorithm with the standard datasets of the Web Service Challenge 2008.

All these contributions are described in the following publication:

> Pablo Rodríguez-Mier[1], Manuel Mucientes[1], Juan Carlos Vidal[1], and Manuel Lama[1]. An Optimal and Complete Algorithm for Automatic Web Service Composition. *International Journal of Web Service Research*, 9(2):1–20, 2012. IGI-GLOBAL. ISSN: 1545-7362. DOI:10.4018/jwsr.2012040101.
> URL: `http://dx.doi.org/10.4018/jwsr.2012040101`.

3.1 Abstract

The ability of web services to build and integrate loosely-coupled systems has attracted a great deal of attention from researchers in the field of the automatic web service composition. The combination of different web services to build complex systems can be carried out using different control structures to coordinate the execution flow and therefore, finding the optimal combination of web services represents a non-trivial search effort. Furthermore, the time restrictions together with the growing number of available services complicate further the composition problem. In this paper we present an optimal and complete algorithm which finds all valid compositions from the point of view of the semantic input-output message structure matching. Given a request, a service dependency graph which represents a suboptimal solution is dynamically generated. Then, the solution is improved using a backward heuristic search based on the A* algorithm which finds all the possible solutions with different number of services and runpath. Moreover, in order to improve the scalability of our approach, a set of dynamic optimization techniques have been included. The proposal has been validated using eight different repositories from the Web Service Challenge 2008, obtaining all optimal solutions with minimal overhead.

3.2 Introduction

Nowadays, Service-Oriented Architectures (SOA) [80] are gaining importance because of the ability to build interoperable services that can be shared over a network within multiple

[1]Centro Singular de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela.

platforms. Thus, companies are starting to apply this principles to their business, allowing them to remain cost effective, flexible and competitive. Applications in SOA are built based on services consumed by clients that are not concerned with the underlying implementation. Specifically, web services are the preferred standard-based way to realize SOA.

Web Services are self-contained modular applications described by a collection of operations that are network-accessible through standardized web protocols, and whose features are defined using a standard XML-based language [7]. One of the advantages of web services is to enable greater and easier integration and interoperability among systems and applications through web service composition. This advantage allows web services to be combined by connecting their inputs and ouputs to create larger services (composite services) whose execution is orchestrated by a set of control structures defined in composition languages like WS-BPEL [103, 121]. Thus, the goal of web service composition is to construct new services from existing web services in order to satisfy a request (basically a set of provided inputs and a set of wanted outputs by the client) which cannot be solved by a single web service. The matching between inputs and outputs can either be done syntactically, using the information described in WSDL [29], or semantically, using semantic markup languages like OWL-S [62] or WSMO [101].

The automatic composition problem may seem trivial problem when there are a limited number of services in a single-service architecture. However, the problem increases in complexity when the goal is to obtain optimal compositions over large web service repositories using different control structures to manage the composition flow. In fact, the web service composition problem can be reduced to the boolean satisfiability problem, i.e., the problem is NP-complete and therefore it cannot be solved in polynomial time [72].

Research in this field has grown rapidly in recent years. Some approaches, such as [45,50, 85, 110, 128] treat the service composition as an artificial intelligence (AI) planning problem, where a sequence of actions lead from a initial state (inputs and preconditions) to a goal state (required outputs). These techniques work well when the repository size is relatively small and the number of constraints is high. However, most of these proposals have some drawbacks: high complexity, high computational cost and inability to maximize the parallel execution of web services.

Other approaches, such as [12, 38, 98] scale better than other techniques when the interactions among services and the number of constraints is huge. Despite being scalable, these techniques do not guarantee to obtain the optimal solution, and also are extremely slow and

memory intensive.

The most recent approaches, such as [41, 44, 48, 54, 75, 107, 124, 125, 129], consider the problem as a graph/tree search problem, where a search algorithm is applied over a sub-optimal graph in order to find a optimal (or near-optimal) solution. These proposals are simpler than the AI planners due, in part, to the use of a smaller number of constraints during the search. However, most of these approaches rely on very complex dependency graphs that have not been optimized to reduce data redundancy. Therefore, the scalability of these algorithms may also be adversely affected when the interaction among services and data is huge due to the redundancy of the repository.

This paper addresses the problem of the web service composition as a graph search problem from the point of view of the semantic input-output message structure matching, i.e, we do not take into consideration the non-functional properties (NFPs). The novelties of our proposal are:

1. The method is able to calculate, given a request, an extended service dependency graph which represents a valid but sub-optimal solution for the request.

2. The heuristic search algorithm, based on the well-known A*, finds all optimal solutions from the point of view of the number of services and execution path (runpath). This, it maximizes the parallel execution of services and minimizes the number of services.

3. We define set of optimizations to reduce the graph size, based on the redundancy analysis and service dominance.

4. We include a method to reduce dynamically the possible paths to explore during the search by filtering equivalent compositions.

We have validated our algorithm with the eight datasets defined by the Web Service Challenge 2008 of the EEE conference [13]. Also we have compared our approach with the results of the participants of the Web Service Challenge 2008.

The rest of the paper is organized as follows: Section 3.3 describes the different approaches that have already been proposed. Section 3.4 introduces the basis of web service composition. Section 3.5 illustrates the proposed A* algorithm for web service composition. Section 3.6 presents some optimization techniques to improve the performance of the algorithm. Section 3.7 analyzes the algorithm with eight different repositories and compares the results with other approaches. Section 3.8 points out the conclusions.

3.3 Related Work

Heuristic algorithms have proved their efficiency in the field of the automatic web service composition. Particularly, the use of graph-based and tree-based search algorithms has been studied before [58, 66] to solve a web service composition in large repositories, showing great results. Although there are similarities among all proposals, they differ in many concepts, such as performance, information handling, graph/tree encoding, solution quality, etc. In this section, a brief analysis of some approaches is presented.

Shiaa et al. [107] present an approach to automatic service composition with semantic matching. Given a request (goals, inputs and outputs), a set of matching services are discovered from the repository, applying semantic matching between service properties and the composition request. Then, a graph is created dynamically by connecting semantically similar nodes (single services) to each other. Once the graph is created, a search over it is performed building acyclic tree structures from goal nodes to start nodes. One major drawback of this proposal is that it does not take into account the use of heuristics in order to speedup the search, so searching for an optimal composition in large repositories may be infeasible. Moreover, there are no experimental results to validate the model.

Kona et al. [54] propose a simple but effective approach for semantic web service composition. In this work, a composition is generated as a directed acyclic graph from a user request. The graph (divided in a set of layers) is calculated iteratively, starting with the input parameters provided by the requester. In each step, all possible services from the repository that can be invoked are added to the current layer. Although the useless services are filtered, the algorithm cannot find an optimal composition. A heuristic search over the graph is required in order to minimize the number of services in the composition.

Yan et al. [129], present an automatic service composition algorithm using AND/OR graph. In this proposal, an AND/OR graph is created from a request, connecting services by their inputs and outputs. Then, a search over the graph is performed using the AO* search algorithm. Although this proposal shows a great performance over large repositories, the algorithm does not guarantee to obtain the optimal compositions from the point of view of the number of services, as can be seen in the results of the competition[2]. Moreover, the authors have not implemented optimization techniques in order to improve the scalability of the algorithm.

[2] http://cec2008.cs.georgetown.edu/wsc08/downloads/WSCResult.pdf

Oh et al. [70,74] propose a Web-Service Planner using the A* search algorithm (WSPR*), an improvement of the WSPR planner, which was at third place in the WSC'08. In this approach, the use of the A* algorithm allows finding an optimal composition based on some heuristic costs. The heuristic function is defined as the set of required parameters found by the algorithm. This heuristic function has an important drawback: it is not able to guide the search when only the last services of a composition produce all the required parameters. On the other hand, the transition function only allows the addition of a single service in each step.

Wu et al. [125] presented AWSP, an automatic web service planner based on heuristic state space search. In this work, an A* is used to search minimal compositions in terms of execution path. The search is performed using different operators which allow the movement from one state to another, adding a new service in each step. This movement can be done either forward or backward, although the last one is clearly better. To do this, two different heuristics were implemented based on a parameter distance defined by the authors. This approach has some drawbacks: firstly, authors do not consider the use of stratified methods previous to the search. These methods allow to quickly reduce the search space size, and can be used in dynamic environments as the computation of service graphs has not an important impact on the overall performance. This, in dynamic environments, where inputs and outputs can change, the recalculation of the graph can be done without affecting too much the performance. In second place, the algorithm cannot manage parallel execution of services. Third, they do not take into account the detection of redundancy, which can seriously affect search performance. Finally, in forth place, more tests are required to confirm the advantages of this approach, comparing it with other similar AI planners as WSPR*.

Benthem et al. [3] got the second place in the Web Service Challenge 2008 with RugCo, an automatic web service compositor. This algorithm uses a tree based search to find compositions that satisfy a request. The search is performed expanding nodes and resolving the new dependencies generated in each step until no more dependencies are discovered. Since during the search a large number of expanded nodes is generated, the authors introduce a heuristic approximation (beam search) to analyze only the most promising nodes. Despite the authors found solutions for the three datasets proposed in the WSC'08, the major drawbacks of this approach are: 1) the beam search does not guarantee to obtain optimal solutions, as only the most promising nodes are expanded, so the algorithm is neither complete nor optimal, 2) the search minimizes the number of services in the composition, but not the execution path and 3) beam search does not scale well with the size of the space search, which implies bad

performance in large datasets.

Weise et al. [124] obtained the forth place in the WSC'08 with an architecture which combines three different algorithms (uninformed search based on ID-DFS, a greedy search and a genetic algorithm [123]). The architecture integrates a module called "Strategy Planner" which decides the best algorithm in each case. The results obtained with this system are not surprising. The ID-DFS is an uninformed search based on the depth-first search (DFS) with iterative deepening (ID). This method is very simple and ineffective to solve a web service composition problem as the time complexity grows exponentially with the depth. When the dataset is too big for the ID-DFS algorithm, the greedy algorithm is used instead of the ID-DFS. This approach is very similar to the DFS, but a heuristic is used to sort the set of candidate nodes to explore. The greedy algorithm works as bad as the ID-DFS in the worst case scenario. On the other hand, a genetic algorithm is used for all those cases where the ID-DFS and the greedy search cannot find a solution. This algorithm uses a set of evolutionary operators to obtain near-optimal compositions minimizing multiple objectives. However, the results obtained in the WSC'08 show the ineffectiveness of this approach. The major drawback of this algorithm is the fitness function. The fitness is measured by calculating two objectives: composition size and number of wanted (unsatisfied) parameters. This evaluation does not work well when the solutions have a long runpath and the last service or services provide all wanted parameters. In this scenario, there are no information about which solution is better until the complete composition is reached, so in each generation, the best individuals are those with a less number of services. This evaluation can prevent the algorithm to find a solution. Moreover, the algorithm is an order of a magnitude slower than the other approaches.

With this state of the art, we can conclude that the main differences between our proposal and other approaches are:

- The construction of a non-redundant service dependency graph at the first stage by removing unused services and combining the equivalent ones. Other approaches use simple filtering techniques that do not remove all data redundancy.
- The use of the A* algorithm backwards, handling multiple services in each step in order to maximize the execution in parallel of the web services.
- The detection of all valid compositions with different number of services and runpath. Other approaches only find an optimal composition with minimum number of services or minimum runpath.

- The use of dynamic optimization during the search, that reduces the number of possible paths to explore by combining equivalent combination of services.

In the following sections we describe in detail the composition problem and how it can be solved with our proposal.

3.4 Web services composition

In order to compose web services, we must define the relationship among services. From a functional point of view, a web service is a software component that receives a set of inputs and generates a set of outputs after the execution. Thus, a web service w can be described by a set of inputs $W_{in} = \{I_1, I_2, ...\}$ and a set of outputs $W_{out} = \{O_1, O_2, ...\}$. Outputs from a service can be provided as inputs to other service only if there is a semantic relationship between them. In our approach, we have modeled this restriction as a hierarchical class/subclass relationship between concepts, so we consider that an output of a service O_{so} matches the input of other service I_{si} when O_{so} is a subclass of I_{si}. In general, when a concept C_i is a subclass of a concept C_j ($C_i \subseteq C_j$), then there is a semantic matching between C_i and C_j.

Another important concept is a web service request. A request R is composed by a set of inputs ($R_{in} = \{I^1_{in}, I^2_{in}, ...\}$) provided by the requester, and a set of outputs ($R_{out} = \{O^1_{out}, O^2_{out}, ...\}$) that the requester expects to obtain. Given a request $R_{user} = \{R_{in}, R_{out}\}$, where $R_{in} = \{I^1_R, I^2_R, ...\}$ and $R_{out} = \{O^1_R, O^2_R, ...\}$, and given a web service $S = \{S_{in}, S_{out}\}$ where $S_{in} = \{I^1_S, I^2_S, ...\}$ and $S_{out} = \{O^1_S, O^2_S, ...\}$, the web service S can be invoked only if $R_{in} \supseteq S_{in}$, i.e., for each input $I_S \in S_{in}$ there exists an input $I_R \in R_{in}$ such that I_R is equal or subclass of I_S ($I_R \subseteq I_S$). Also, R_{out} will be satisfied only if $R_{out} \subseteq S_{out}$, i.e., for each output $O_R \in R_{out}$ there exists an output $O_S \in S_{out}$ such that O_S is equal or subclass of O_R ($O_S \subseteq O_R$).

Considering this description for web services, the composition problem can be formulated as the automatic construction of a workflow that coordinates the execution of a set of services that interact among them through their inputs and outputs (applying the semantic matching). This workflow, therefore, has services and a set of control structures that define both the behavior of the execution flow and the inputs/outputs of the services related to those structures. Despite the amount of different control structures defined in composition languages like WS-BPEL, we take into account only two of the most important ones: sequence and split. These structures allow to build most of the possible compositions and they work as follows:

- Sequence structure: the output of a service is the input of one of the following services of the sequence. This is the basic control structure of the workflow languages.
- Parallel (split): two or more services are executed in parallel and, as result, produce several and different outputs.

Regarding to the complexity analysis of the search space, the number of combinations to be analyzed using a brute-force algorithm grows very fast. To demonstrate this, we can assume that, given a service, each of its inputs is provided by a different service (worst case). The complexity in this scenario is $O(m^{nd})$, where m is the average number of services in the repository that generate the same output, n is the average number of inputs from web services and d is the depth at which all inputs are resolved. Since there are m services that provide each required input, the number of possible choices in order to resolve all inputs from a service is m^n. Each of these combinations represent a set of services executed in parallel, that can be expanded again. Fig. 3.1 shows the size of the search space for different values of the runpath ($d = 1...5$), with $n = 1$, $n = 2$, $n = 3$ (one, two and three inputs respectively for each service in repository) and $m = 5$ (5 services per output on average).

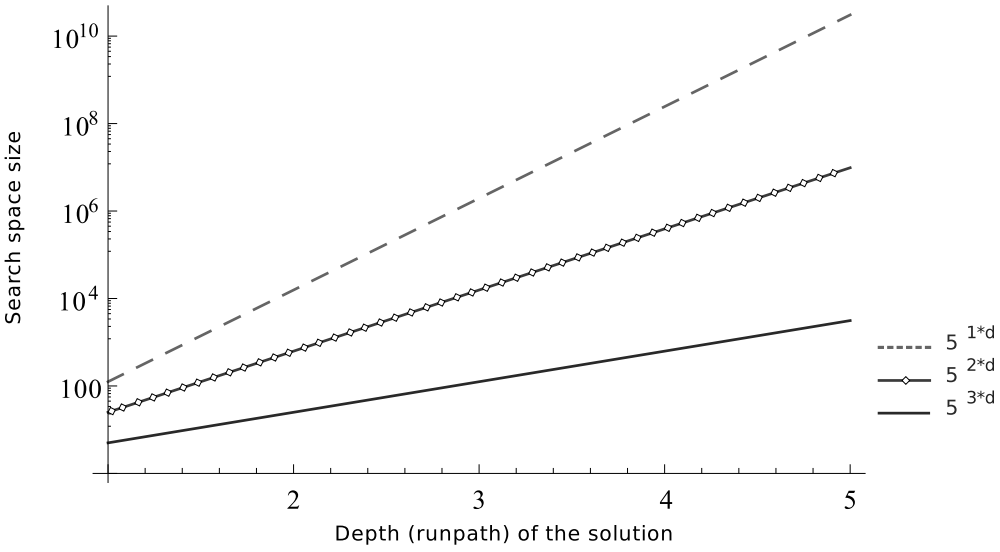

Figure 3.1: Search space size for n=1, n=2, n=3 (1, 2 and 3 inputs per service) and m=5 (5 services per output on average) with variable runpath.

As can be seen, this kind of compositions have an exponential growth of paths to explore. The search space size in the case of a repository of services with three inputs on average (n) and four possible choices to provide an input to a service (m), where the solution has a runpath of 10 (i.e, d=10 splits connected in sequence), reaches the value of $5^{3 \cdot 10} = 9.3132 \times 10^{20}$ possible paths to explore. Given the large number of combinations, the problem of searching an optimal execution path is not trivial, and it is therefore necessary to reduce the number of combinations. In order to reduce the search space size, our algorithm includes some optimization techniques, which are described in Sec. 3.6.

3.5 A* algorithm for web services composition

As previously discussed, given the large number of possible paths to explore, a fast algorithm is required in order to find an optimal solution in a reasonable period of time. Although the high space complexity makes the use of traditional search algorithms unpractical for large repositories, the problem can be solved by using a good heuristic in the search and applying some optimization techniques and data preprocessing.

The A* algorithm, developed by Hart et al. [40], is one of the most popular pathfinding algorithms. This algorithm uses a heuristic function $h(n)$ to estimate the cost from the current node to a goal node, and a function $g(n)$ to calculate the cost from the starting node to the current node. Therefore, the path cost is defined as $f(n) = g(n) + h(n)$. Choosing a good h function has an important impact on the search process. The better this function is, the faster the solution will become. However, there is a restriction on it: h cannot overestimate the cost to reach the goal, otherwise, the algorithm could find a solution with higher cost than the optimal one.

Our proposal, based on A* algorithm, follows the next steps: first, a web service dependency graph is computed (Sec. 3.5.1). Then, a reduction on the number of services is performed by eliminating unused services and combining equivalent services (Sec. 3.6). Finally, the A* search is applied over the reduced graph, which finds all optimal service compositions, with minimum number of services and execution path (Sec. 3.5). These steps will be described in the following sections.

3.5.1 Extended web service dependency graph

Web services composition requires the combination of many atomic services that can be executed in sequence or in parallel as previously mentioned. Given a service request, an extended service dependency graph (SDG) with a subset of the original services from an external repository is dynamically generated. This subset contains the solutions that meet the request and consists of a set of layered services (splits) connected in sequence. Each layer contains all services from the repository that can be executed with the outputs of the previous one. Fig. 3.2 shows an example of a SDG with i layers and n services in each layer. The expression for a layer can be defined as follows:

$$L_i = \{S_i : S_i \notin L_j(j < i) \wedge I_{Si} \cap O_{i-1} \neq \emptyset \wedge I_{Si} \subseteq I_R \cup O_0 \cup \ldots \cup O_{i-1}\}$$

where, for each layer L_i:

- S_i is a service on the *i-th* layer.
- O_i is the set of outputs generated in the *i-th* layer.
- I_{S_i} is the set of inputs required for the execution of service S_i.
- I_R is the set of inputs provided by the requester.

The construction of the graph can be done in a simple manner. Alg. 1 explains with pseudocode the construction of the graph iteratively. Lines 1-5 initialize the variables used throughout the algorithm: *newOutputs* (outputs generated in the last layer that have not been generated previously), I_a (available inputs for the current layer), i (current layer) and *Layers* (set of all generated layers). Note that *newOutputs* and I_a are initialized with the same value I_R, as the provided inputs are the first available inputs to the composition and have not been used yet by any service. The main loop starts at line 6. Inside this loop, each layer is calculated following these steps:

1. Obtain all outputs from the previous layers. This outputs are the available inputs to the current layer (L. 8-10).
2. For each service in the repository:
    a) Check if the service has not appeared in previous layers (L. 13).

b) Check if the service can be invoked (i.e. receives all its inputs from previous layers) (L.14).

c) Check if the service uses at least one output that has not been used previously (L. 15).

d) If (a), (b) and (c) are true, then the service is added to the current layer.

3. If the available inputs to this layer contain the wanted outputs (solution reached) and the previous layer produces at least one of the wanted outputs, then a dummy service (R_o^n) is added to the current layer. All R_o^n services are the initial nodes of the search (each initial node will lead to a solution with different runpath). (L. 20-25)

4. Once all services are selected for the *i-th* layer, *newOutputs* is updated by adding the outputs of the *i-th* layer and deleting the outputs generated in previous layers. Note that with this operation, only the outputs that have not been used before will remain for the next iteration (L. 26).

In order to speed up the calculation of the graph, we used a pre-computed table that maps each input to the services that use it. Thus, for each output generated in a layer, we can obtain all possible services for the next layer very quickly. Fig. 3.3 shows an example of a service dependency graph with five layers and two different solutions. The dark gray services correspond with the services of the solution with the largest runpath (the first and the last layers are not computed for the runpath). R_i, R_o^1 and R_o^2 are dummy services. R_i is a service which provides the requested inputs, R_o^1 is a service which uses the requested outputs (so there is a solution with a runpath of 2) and R_o^2 is a service which uses the requested outputs but in layer 4 (runpath of 3). Thus, in this example, two different solutions for the same request can be observed: $Sequence(R_i, Split(S_{1,2}, S_{2,3}), S_{2,2}, R_o^1)$ and $Sequence(R_i, S_{1,1}, S_{2,1}, Split(S_{3,1}, S_{3,2}), R_o^2)$.

Generally, stratified methods like this have a high performance, and allow to reduce the total search space easily, as some constraints (in this case, inputs and outputs) are exploited to reduce the number of services that can be used. These methods work well in static environments, where the service information does not change. In real word, where the inputs and outputs, service availability and other parameters may change, these methods must be adapted. Basically, to ensure the validity in dynamic environments, a fast check can be done while the algorithm is searching for a solution. If any change is detected on any of the services

Algorithm 1 Extended service dependency graph algorithm

```
1:  newOutputs := I_R
2:  I_a := I_R
3:  i := 0
4:  Layers := ∅
5:  n := 0
6:  repeat
7:      L_i := ∅
8:      for L_j (j < i) do
9:          I_a := I_a ∪ Outputs{L_j}
10:     for Service S_i ∈ Repository do
11:         O_s := Outputs{S_i}
12:         isNewService := S_i ∉ L_j(j < i)
13:         hasInputsAvailable := Inputs{S_i} ⊆ I_a
14:         usesNewInputs := Inputs{S_i} ∩ newOutputs ≠ ∅
15:         if isNewService ∧ hasInputsAvailable ∧ usesNewInputs then
16:             L_i := L_i ∪ S_i
17:     if I_a ⊇ wantedOutputs ∧ newOutputs ∩ wantedOutputs ≠ ∅ then
18:         R_o^n.inputs = wantedOutputs
19:         R_o^n.outputs = ∅
20:         L_i := L_i ∪ R_o^n
21:         n := n + 1
22:     newOutputs := newOutputs ∪ O_s − I_a
23:     Layers := Layers ∪ L_i
24:     i = i + 1
25: until L_i ≠ ∅
```

selected by the search algorithm, the service dependency graph must be recalculated starting from the layer which contains the service. Specifically, two situations may occur:

- A service is not accessible: given that our algorithm finds all possible solutions, when a service becomes unavailable, the solutions which contain the unavailable service must be discarded. The other solutions will be still valid.

- A new service is available: in this case, the service dependency graph must be partially rebuilt starting from layer L_{i+1}, where L_i is the layer at which the inputs required by the new service are provided (i.e., the layer which contains the service, according to the definition of L_i defined before).

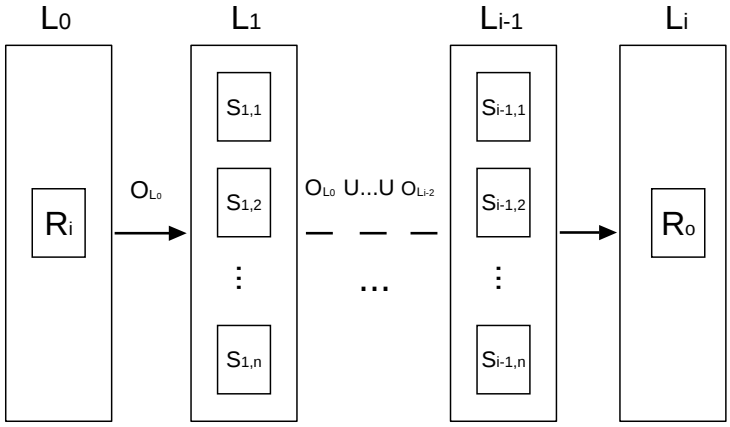

Figure 3.2: Example of i layers, with n services per layer

3.5.2 A* algorithm description

Once the graph is calculated, a search over it must be performed. The search algorithm will traverse the graph backwards, from the solution (the service whose inputs are the outputs wanted by the requester), to the initial node (the service whose outputs are the provided inputs). As mentioned before, our heuristic algorithm is based on an implementation of the A* heuristic search. There are three principal concepts in this type of algorithms: the neighborhood function, the cost function and the heuristic function.

In order to perform the search process, the search space must be divided into nodes. Each node will contain a set of services from a graph layer that can be executed in parallel. Thus, a path will be composed of a list of neighbor nodes, which represents the sequential execution of the path. Thus, the starting node will only contain the service labeled as R_0 in Fig. 3.2. This service represents the outputs wanted by the requester, as their inputs match with them. To generate all possible neighbors from a node, the following steps are performed:

1. Calculate, for each input of a node, a list of services from the previous layer that provide it. If there are no services in the previous layer for that input, a dummy service that generates this input and receives the same input is created. This dummy holds the dependency so it can be resolved later.

2. Make all combinations among services from each list. These combinations will generate all possible neighbors from the current node.

3. Remove all equivalent neighbors. This process will be described in Sec. 3.6.

For example, given a node N with a service S in the layer L_i, with $I_s = \{a,b\}$ and a set of services X,Y,Z in the layer L_{i-1} where $O_x = \{a\}$, $O_y = \{b\}$ and $O_z = \{a,b\}$, we construct a list of services for each input of S:

- $Set(a) = \{X,Z\}$
- $Set(b) = \{Y,Z\}$

Then, we generate all combinations. Each combination will constitute a neighbor node from N. The possible combinations are: (X,Y), (X,Z), (Y,Z), (Z). All these nodes generate all the required inputs for node N (a, b).

On the other hand, the behavior of the A* algorithm depends on two functions: $g(N)$, the cost, and $h(N)$, the heuristic. N is a composite service obtained as a path over a set of nodes (N_i), where N_i is the set of services in layer L_i. One of the goals is to minimize the number of web services in a composition, therefore, the cost function should calculate the length of a composition based on the number of services. On this basis, we define a function $g(N)$ as:

$$g(N) = \sum_{i=L_N}^{\#L} cost(N_i) \tag{3.1}$$

where L_N is the first layer of the current composition service, $\#L$ is the number of layers and *cost* is a function that retrieves the number of services from node N_i. The dummy services in a node will not contribute to this cost.

The other function is the heuristic. This function should estimate the cost to the solution. A good choice is to use, as heuristic, the layer in which the node is located. The layer number indicates the distance to the initial node. Thus, a service in layer 3 means that the algorithm needs three more steps in order to reach the start node. The heuristic function is defined as:

$$h(N) = distance(N_i) \tag{3.2}$$

Putting (3.1) and (3.2) together, function $f(n)$ is defined as (3.3):

$$f(N) = \sum cost(N_i) + distance(N_i) \tag{3.3}$$

Figure 3.3 shows an example of a minimum composition path detected with this algorithm. In the next section, a set of optimization techniques are explained.

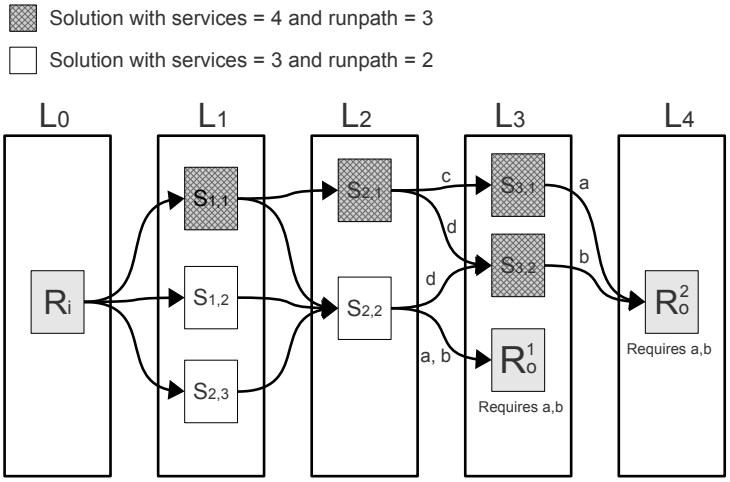

Figure 3.3: Example of two solutions with different runpath and different number of services

3.6 Optimization techniques

In order to achieve a significant performance improvement on the search process, we designed two techniques that reduce the number of possible paths to explore: *Offline Service Compression* and *Online Node Reduction*.

3.6.1 Offline Service Compression

The essence of this technique is to replace equivalent services from each layer in the graph by the representative service, which implies a lower number of paths to explore during the search. This process is subdivided into two steps: remove unused services and detect equivalent services. These steps are described below:

- Remove unused services:

    1. Create an empty list M. This list will contain all the required inputs to get the solution.
    2. Create an empty list U. This list will contain all unused services.
    3. Traverse backwards the graph, starting from the final layer.

4. For each layer L_i in the graph:
   a) Create an empty list R. This list will contain all the required inputs for this layer.
   b) For each service S in the current layer:
      i. Check if $O_s \subseteq M$, where O_s are the service outputs. If M is empty, skip this step.
      ii. If S meets the condition or M is empty, add all inputs from S to the list R.
      iii. In other case, add S to the list U
   c) Add all inputs from R to the list M.
5. Finally, remove from the graph, each service in U

- Detect and combine equivalent services. For each layer in the graph:
  1. Group services by the equivalence of their inputs. Two services have equivalent inputs if the services from the graph that provide their inputs are the same.
  2. For each group:
     a) Check if $S_i \succeq S_j$ for each service S_i and S_j from a group.
     b) If S_i meets the previous restrictions, then select S_j as the representative service. S_j must be deleted.

One service S_i with parameters $P_{S_i} = \{P^1_{S_i}, P^2_{S_i}, ..., P^n_{S_i}\}$ dominates other service S_j ($S_i \succeq S_j$) with parameters $P_{S_j} = \{P^1_{S_j}, P^2_{S_j}, ..., P^n_{S_j}\}$ if:

$$\forall\, k \in \{1,\ldots,n\}\ P^k_{S_i} \geq P^k_{S_j} \wedge \exists\, k \in \{1,\ldots,n\}, P^k_{S_i} > P^k_{S_j}$$

In our case, we consider only the outputs of a service S_i (O_{S_i}) as the single parameter of S_i. The inputs are not considered as the services are grouped by the equivalence of their inputs. To clarify this point, the dominance between two services S_i and S_j with outputs O_{S_i} and O_{S_j} respectively can be done as follows:

1. Set $List_i$ as the list of services from the graph such that their inputs are a subset of O_{S_i}.
2. Set $List_j$ as the list of services from the graph such that their inputs are a subset of O_{S_j}.
3. Compare both lists. If $List_i \supseteq List_j$ then go to the next step. Else, the restriction is not met and therefore S_i and S_j cannot be combined.

4. Check if O_{S_i} resolves the same or more inputs from each common service than O_{S_j}. For example, if $O_{S_i} = \{a,b\}$ and $O_{S_j} = \{a,c\}$, and $List_i = List_j = X(a,b,c), Y(a,c)$, where $X(a,b,c)$ and $Y(a,c)$ are services that receive as inputs (a,b,c) and (a,c) respectively, we must verify which inputs are resolved with O_{S_i} and O_{S_j}. So, in this example, O_{S_i} resolves input a,b from X and a from Y, and O_{S_j} resolves a from X and a,c from Y. Therefore, $S_i \not\succeq S_j$.

This technique can be used in both static and dynamic environments. Suppose that the service S, which generates the outputs a and b ($S \rightarrow (a,b)$) is the representative service of the group which contains the services $U \rightarrow (a)$ and $V \rightarrow (b)$. If the service S becomes unavailable, then the services U and V can be selected to replace the representative service. The generation of the all possible replacements can be done in the same way as the calculation of the neighborhood of a node, as explained in Section 3.5.

3.6.2 Online Node Reduction

This technique consists in the combination of equivalent neighbors during the A* search process. Given that a node can generate equivalent neighbors (different combination of services that together are equivalent), a mechanism to delete this type of redundancy must be implemented. Two nodes are equivalent if they meet two conditions:

1. Neighbors from the node must have the same $f(n)$ value.

2. Services from graph that provide the inputs required for each neighbor must be the same.

The first condition is obvious: two neighbors cannot be reduced if the $f(n)$ value is different, as they will generate different paths to the solution. The second condition refers to the equivalence of the inputs. As before, a list of services that provides the required input for each neighbor must be calculated and then compared. Only nodes with same lists of services and $f(n)$ value can be combined. This technique is performed while the neighbors are being generated.

3.7 Experiments

Our analysis consists in two parts: (1) we validate the algorithm with eight different repositories from Web Service Challenge 2008 and (2) we measure the speed up obtained with the optimization techniques.

3.7.1 Web Service Challenge 2008 Datasets

In order to evaluate the correctness and the performance of our algorithm in different situations, we have carried out some experiments[3] using eight public repositories from Web Service Challenge 2008[4]. These repositories contain from 158 to 8119 services defined using WSDL. Also, inputs and outputs are semantically described in a XML file. Although there are other benchmark datasets for automatic web service composition [71], the most efficient algorithms have been evaluated using the WSC datasets.

Table 3.1: Characteristics of the Web Service Challenge repositories.

Test	#Services	#Inputs	#Outputs
WSC'01	158	735	778
WSC'02	558	2,972	2,890
WSC'03	604	3,254	3,129
WSC'04	1,041	5,781	5,611
WSC'05	1,090	5,816	5,953
WSC'06	2,198	12,218	11,831
WSC'07	4,113	22,324	22,392
WSC'08	8,119	44,569	44,628

Table 3.1 shows in detail the characteristics of each dataset. The first column indicates the number of services in the repository (#Services). As can be seen, the number of services is variable and enough for a full validation. Table also shows the total number of inputs (#Inputs) and the total number of outputs (#Outputs).

[3] An online application is available to test our algorithm with the same datasets used in this experiments: http://citius.usc.es/wiki/inv:composit

[4] http://cec2008.cs.georgetown.edu/wsc08/downloads/ChallengeResults.rar

The solutions provided by the WSC'08 are showed in Table 3.2. Column '#Services" indicates the number of services for the shortest[5] solution (in number of services). Column "exec. path" shows the runpath for that solution. Finally, column "#Solutions" indicates the number of different solutions for that dataset.

Table 3.2: Web Service Challenge: Solutions provided by the WSC'08

Test	#Services	Exec. path	#Solutions
WSC'01	10	3	3
WSC'02	5	3	4
WSC'03	40	23	1
WSC'04	10	5	2
WSC'05	20	8	2
WSC'06	40	9	2
WSC'07	20	12	2
WSC'08	30	20	2

Table 3.3: Algorithm results for the eight datasets

			Solution with min. Services				Solution with min. Runpath			
Test	Gr.serv	#Sol.	Iter.	Time(ms)	#Serv	Runpath	Iter.	Time(ms)	#Serv	Runpath
WSC'08-1	46	7	25	81	10	3	25	81	10	3
WSC'08-2	45	4	9	147	5	3	9	147	5	3
WSC'08-3	42	1	24	436	40	23	24	436	40	23
WSC'08-4	27	2	11	101	10	5	11	101	10	5
WSC'08-5	72	6	69	487	20	8	69	487	20	8
WSC'08-6	132	12	115	3,306	35	14	126	3,508	42	7
WSC'08-7	110	2	33	3,345	20	12	33	3,345	20	12
WSC'08-8	78	3	128	3,608	30	20	128	3,608	30	20

3.7.2 Results

Our algorithm was implemented using Java™ JDK 1.6 and tested with Java™ SE build 1.6.0_22-b04 64-bit. All the experiments were performed under an Ubuntu 64-bit server

[5]Note that these values are only indicative. Smaller values have been found by our algorithm and by other participants

Table 3.4: Comparison with the participants of the WSC'08

	Tsinghua		Groningen		Pennsylvania		Kassel		**USC**	
	Result	Points	Result	Points	Result	Points	Result	Points	**Result**	**Points**
WSC'08-4										
Min services	10	6	10	6	10	6	10	6	**10**	**6**
Min execution	5	6	5	6	5	6	5	6	**5**	**6**
Time (ms)	312	2	219	4	28,078	0	828	0	**101**	**6**
WSC'08-5										
Min services	20	6	20	6	20	6	21	0	**20**	**6**
Min execution	8	6	10	0	8	6	8	6	**8**	**6**
Time (ms)	250	6	14,734	2	726,078	0	300,219	0	**487**	**4**
WSC'08-6										
Min services	46	0	37	6	-	0	-	0	**42/35**	**0/6**
Min execution	7	6	17	0	-	0	-	0	**7/14**	**6/0**
Time (ms)	406	6	241,672	2	-	0	-	0	**3,508/3,306**	**4/4**
TOTAL		44		32		24		18		**44**

workstation (kernel 2.6.32-27) with 2.93GHz Intel® Xeon® X5670 and 16GB RAM DDR-3. Table 3.3 shows the results obtained with a minimum runpath and a minimum number of services. This table is organized as follows: the first column indicates the dataset name. The second column indicates the number of services in the service dependency graph (including dummy services). "#Sol" represents the number of solutions obtained by our algorithm, and "Iter." indicates the number of steps executed by the A* search algorithm until the solution was reached. "Time" is the elapsed time until a solution was found (including the time spent in the generation of the service dependency graph), while "#Serv." indicates the number of services obtained by the algorithm. Finally, "runpath" represents the length of the execution path of the solution. Columns 8-11 have the same meaning as columns 4-7 but for the solutions with minimum runpath.

As can be seen, in all cases (except in WSC'08-6) the solution with minimum number of services is the solution with minimum runpath too. The first thing that must be noticed is that the solutions obtained by our algorithm are the best for all datasets (according to the solutions provided by WSC'08, see Table 3.2), except in the case of the dataset WSC'08-6, where our algorithm finds a solution with lower number of services (35 vs 40) and a solution with shorter

runpath (7 vs 10). Our approach also scales well with the number of services (3,345 ms for the dataset with 4,113 services and 3,608 ms for the dataset with 8,119 services).

Moreover, *the algorithm finds all possible solutions* (column "#Sol.") for all datasets, showing a great performance as in all cases the bests solutions were found in a very short period of time. This feature is an important advantage over the other approximations since it shows that is possible to compose services automatically using an optimal and complete algorithm.

3.7.3 Comparison

In order to prove the validity of our approach, a comparison with the participants of the challenge has been done, following the rules defined by the WSC'08[6]. The quality of each composition is measured using three parameters (number of services, runpath and time) in accordance with the scoring rules as follows:

- +6 Points for finding the minimum set (Min. Services) of services that solves the challenge.
- +6 Points for finding the composition with the minimum execution length (Min. Execution) that solves the challenge.
  - +6 Points for the composition system which finds the minimum set of services or execution steps that solves the challenge in the fastest time (Time (ms)).
  - +4 Points for the composition system which solves the challenge in the second fastest time.
  - +2 Points for the composition system which solves the challenge in the third fastest time.

As can be seen, these rules are conflicting, given that some solutions have the minimum runpath but not the minimum number of services. For example, in the WSC'06 dataset, the solution with the minimum number of services (35 services) has a runpath of 14. On the other side, the solution with the minimum runpath has 42 services. With these rules, both solutions obtain 6 points, as the first one has the minimum number of services and the second one the minimum runpath. Despite our algorithm finds both solutions, only one solution is

[6] http://cec2008.cs.georgetown.edu/wsc08/downloads/WSCResult.pdf

taken into account. Thus, our algorithm is clearly penalized by this rating. Regardless of this disadvantage, our algorithm obtained 44 points, the same score as the winners. Note that the time has not been measured under the same conditions because the source code of the other participants was not available. Therefore, the objective criteria for the comparative analysis should be only the number of services and the runpath.

If we compare the quality of the solutions, our algorithm finds better solutions than the other approaches. As can be seen in Table 3.3, the result with the minimum runpath for the dataset WSC'08-6 obtained by our algorithm has 42 services, while the University of Tsinghua obtained a solution with 46 services and the same runpath. On the other hand, if we compare the solution with the minimum number of services, our algorithm finds a composition with 35 services and a runpath of 14, which is clearly better than that provided by the University of Groningen with 37 services and a runpath of 17.

3.7.4 Optimization effect

All the above experiments were performed using all the optimization techniques described on Section 3.6. In this section, we compare the effect of the optimization over the global performance on each dataset, and it is divided into three parts: (1) performance using *offline service compression*; (2) performance using *online node reduction* and (3) performance improvement with both optimizations.

Offline service compression

the results are presented in Table 3.5. As can be seen, the average compression obtained over the graph using "Offline service compression" was close to 40%. The other columns show the average inputs per service, the average outputs per service and the average number of available services in the service dependency graph that provides the same output (with and without optimization). This values can be used to estimate the complexity for each dataset, as explained in Sec. 3.4. Note that the number of services per output decreases as the compression ratio increases (Column 11). This ratio has an important effect on the search performance. More specifically, a worse performance occurs when the number of available services per output is high, since the generation of neighbors in each step is slower. Despite the reduction obtained over the graph size, the complexity of the repository 6 still remained too high, so the algorithm cannot find a solution in a reasonable period of time (all tests were executed using a time limit of 5 minutes).

Table 3.5: Complexity of the Service Dependency Graph (SDG) with and without using Offline Service Compression

	SDG Services			Avg inputs/service		Avg outputs/service		Avg services/input		
	Non-opt.	Opt.	% Compr.	Non-opt.	Opt.	Non-opt.	Opt.	Non-opt.	Opt.	% Compr.
WSC'01	64	46	28.13	3.35	3.15	4.09	4.06	2.09	1.32	36.84
WSC'02	67	45	32.84	3.23	3.17	4.05	4.02	3.19	1.66	47.96
WSC'03	107	42	60.75	3.84	3.95	4.04	4.23	3.80	1.00	73.68
WSC'04	46	27	41.30	4.69	4.66	4.28	4.25	5.20	2.05	60.58
WSC'05	106	72	32.08	3.25	3.11	4.56	4.68	2.43	1.39	42.80
WSC'06	208	132	36.54	5.77	5.87	4.31	4.59	3.83	2.10	45.17
WSC'07	166	110	33.73	3.12	3.05	4.54	4.85	4.47	2.32	48.10
WSC'08	134	78	41.79	3.60	3.56	4.25	4.57	2.61	1.24	52.49
Average			38.39							50.95

Table 3.6: Performance of the algorithm using different optimizations

	No opt. (ms)	Offline service comp. (ms)	Online node reduction (ms)	All opt. (ms)
WSC'01	98.09	82.91	83.96	81.56
WSC'02	157.59	148.03	152.33	147.36
WSC'03	552.92	432.37	438.17	436.48
WSC'04	118.97	115.77	103.26	101.73
WSC'05	1,930.84	490.67	511.28	487.66
WSC'06	∞	∞	23,704.44	3,306.36
WSC'07	3,476.31	3,377.66	3,363.01	3,344.28
WSC'08	5,117.71	3,609.55	3,598.46	3,608.58

Online node reduction

This technique reports a large improvement in performance, as the algorithm obtains solutions in all repositories, including the WSC 2008-6 (23,704 ms, see Table 3.6). In most cases, this method obtains at least the same performance as the offline service compression. Table 3.6 shows in detail the time obtained for each dataset and using different optimizations. Column 2 indicates the time needed to get the solution with the minimum number of services without any optimization. Columns 3, 4 and 5 show the same information but using different techniques. Note that "Offline Service Compression" is not enough to obtain a solution in the dataset

Table 3.7: Speedup obtained with the different optimizations

	Offline service compr.	Online node reduction	All optimizations
WSC'01	18 %	17 %	20 %
WSC'02	6 %	3 %	7 %
WSC'03	28 %	26 %	27 %
WSC'04	3 %	15 %	17 %
WSC'05	494 %	478 %	496 %
WSC'06	0,00 %	∞	∞
WSC'07	3 %	3 %	4 %
WSC'08	42 %	42 %	42 %

WSC'08-6.

Both optimizations

After applying both techniques, our algorithm is able to solve the eight datasets showing a good performance. Table 3.7 shows the percentage of optimization obtained with the different techniques.

In Figure 3.4, we compare the speedup[7] obtained with each optimization over the non-optimized algorithm. Note that with all optimizations, the speedup is over 1.0x, i.e., there is a substantial performance improvement. The improvement on the WSC'06 dataset cannot be measured as there are no results without optimizations, but a comparison can be done using only the results obtained with "Online node reduction" and "All optimizations". For this case, using the values in Table 3.6, we obtain a speedup of 7x with all optimizations (23,704 ms vs 3,306 ms). This is due to the large number of equivalent combinations of services (neighbor nodes) that can be generated in each step.

3.8 Conclusions

In this paper we have presented a complete and optimal algorithm for automatic web service composition based on a heuristic search over a services graph. The graph has been optimized applying different techniques that reduce useless and equivalent services. The pro-

[7]The speedup is calculated as the division of the non-optimized result by the optimized result. Thus, a speedup of 2.0x indicates that the optimized result is two times faster than the non-optimized one.

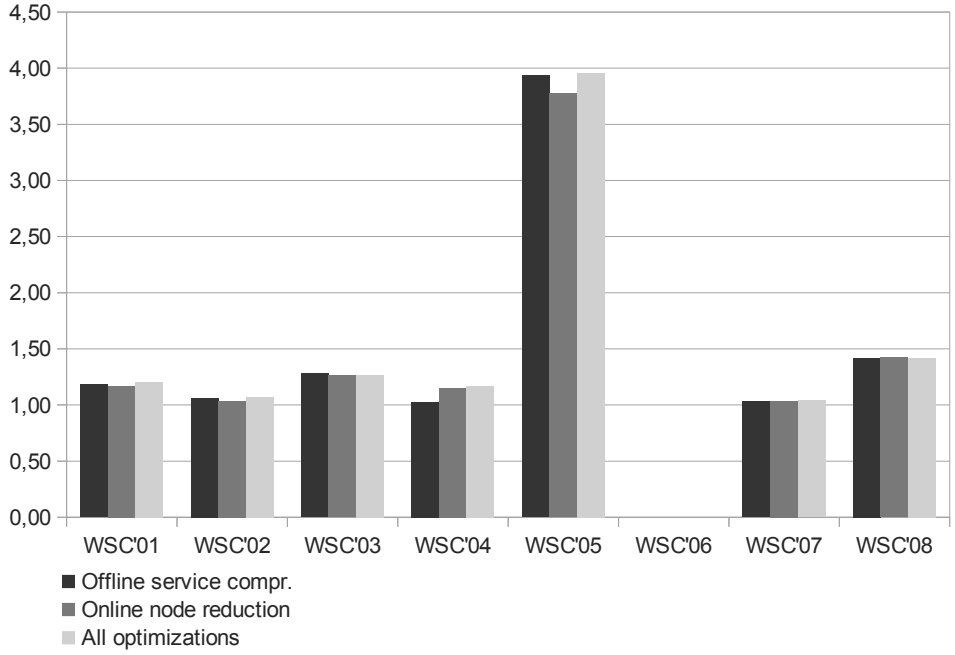

Figure 3.4: Speedup with different optimizations

posed A*-based composition algorithm is executed over the reduced graph using dynamic node reduction and a cost function, which minimizes the number of services and maximizes the parallelization. Moreover, a full validation has been done using eight different repositories from Web Service Challenge 2008, showing a good performance as in all the tests the best solutions, regarding the number of services and runpath, were always found. Also, our algorithm is able to find all the existing solutions. This is not fulfilled by the other algorithms of the WSC'08.

As future work we plan to extend our algorithm by including non-functional properties in our model, such as cost, reliability, throughput, etc. Quality of Service (QoS) characteristics are important criteria for building real world compositions. Our algorithm can be easily adapted to handle these features.

Chapter 4

An Integrated Semantic Web Service Discovery and Composition Framework

As mentioned in Chapter 1, the discovery of services is a fundamental activity that needs to be carried out in service composition. Yet, despite being two interrelated tasks, most of the research in automatic service discovery and automatic service composition has evolved independently. This has led to: 1) interfaces exposed by state-of-the-art discovery engines are not adequate for service composition and 2) composition engines usually assume that services candidates are in place and can be indexed and processed in memory. As a result, there is a lack of integrated approaches that consider the performance and the scalability of the overall integrated system as well as the impact of the discovery in terms of response time during the automatic composition task. With the aim of overcoming these limitations, and based on the research presented in chapter 3, we developed a graph-based framework that integrates both service discovery and optimal service composition. The formal framework presented in this chapter provides a theoretical analysis of graph-based service composition in terms of its dependency with a service discovery by means of a fine-grained I/O discovery interface which reduces the performance overhead without assuming the local availability and in-memory preloading of service registries. We also provide a reference implementation of this formal framework based on the adaptation of two independently developed components, namely ComposIT and iServe, respectively in charge of service composition and discovery.

This reference implementation has been used to empirically study the impact of the discovery task in the whole composition using different optimization mechanisms with varying performance.

All these contributions are encompassed in the following publication:

Pablo Rodríguez-Mier[1], Carlos Pedrinaci[2], Manuel Lama[1], and Manuel Mucientes[1]. An Integrated Semantic Web Service Discovery and Composition Framework. *IEEE Transactions on Services Computing*, 2015. IEEE. ISSN: 1939-1374. DOI: 10.1109/TSC.2015.2402679.
URL: http://dx.doi.org/10.1109/TSC.2015.2402679.

4.1 Abstract

In this paper we present a theoretical analysis of graph-based service composition in terms of its dependency with service discovery. Driven by this analysis we define a composition framework by means of integration with fine-grained I/O service discovery that enables the generation of a graph-based composition which contains the set of services that are semantically relevant for an input-output request. The proposed framework also includes an optimal composition search algorithm to extract the best composition from the graph minimising the length and the number of services, and different graph optimisations to improve the scalability of the system. A practical implementation used for the empirical analysis is also provided. This analysis proves the scalability and flexibility of our proposal and provides insights on how integrated composition systems can be designed in order to achieve good performance in real scenarios for the Web.

4.2 Introduction

Service discovery and composition are in general complex tasks that require considerable effort, especially when vast amounts of services are available. Service discovery solutions range from the initial UDDI proposal that relied on the syntactic description of services and a prefixed categorisation [7], to more advanced generic solutions able to discover Web APIs and

[1]Centro Singular de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela.
[2]Knowledge Media Institute (KMi), The Open University, Milton Keynes, UK.

Web services across domains exploiting rich user-provided semantic service descriptions [81]. Similarly, a plethora of service composition solutions have been produced spanning from mere graphical support to completely automated solutions [36, 92, 113]. Both discovery and composition engines essentially rely on the processing of service descriptions, which increasingly go beyond syntactic representations to include the semantics of the service(s) to enable more advanced computations [64, 77].

An analysis of the service composition literature highlights that, regardless of the approach, a central task that needs to be frequently performed throughout the composition activity, is the discovery of suitable services to use. Whether one looks at fully automated composition engines based on Artificial Intelligence (AI) planning techniques [24, 51, 110], or at more constrained solutions that rely on pre-defined skeletal plans [63, 109], or at graph based approaches focused on semantic input-output parameter matching [3, 44, 54, 68, 70, 89, 107, 129], service discovery is a central activity that needs to be carried out at every main step during the generation of the composition. Yet, despite the strong dependency between both activities, research and development in both areas has evolved for the most part independently.

On the one hand, service discovery has traditionally been approached as a one-of activity to be sporadically carried out by humans when looking for services. As a consequence the interface exposed by discovery engines assumes that requests are fully specified in terms of a well-defined interface and categorisation. Moreover, response times of discovery engines are orders of magnitude above what would be acceptable for a composition engine that should it delegate the thousands discovery requests it needs to issue at composition time [49]. These limitations hamper the development of fast composition systems where discovery and composition are two fundamental, interrelated activities.

On the other hand, partly due to the particularly demanding computational needs of composition algorithms, most composition engines reimplement locally their own discovery methods instead of integrating existing components providing state of the art discovery algorithms. Additionally, this approach relies on the unnecessary and often unrealistic assumption that the entire set of services should be locally available to the composition engine. This assumption requires pre-importing all services locally which is only viable for those registries providing entire public dumps of the service descriptions they hold. Furthermore, most composition engines do not introduce optimisation techniques to improve the scalability by identifying equivalent or dominant functionality that could appear when many differents service registries are involved in the composition. This prevents the use of optimal search strategies since the

complexity usually grows exponentially with the number of services.

In order to tackle the previous problems, a composition framework should consider the following characteristics: 1) provide convenient *fine-grained discovery* mechanisms that could help to discover services able to consume or produce (a subset of) certain types of data as usually required during composition; 2) improve the *response time* of service discovery to process requests very fast; 3) support the integration of *third party service registries* as a key activity in the composition phase; 4) incorporate *optimizations* to improve the scalability of the overall composition process; and 5) find *optimal service compositions* by minimizing different criteria such as the number of services or the length of the composition to avoid complex, unmanageable solutions.

In this paper we present a graph-based framework focused on the semantic input-output parameter matching of services' interfaces that efficiently integrates the automatic service composition and semantic service discovery. The provided framework takes into account all the characteristics indicated in the above paragraph. Notably, the main contributions are:

1. A formal framework that presents a theoretical analysis of graph-based service composition in terms of its dependency with a service discovery and we provide a fine-grained I/O discovery interface which reduces the performance overhead without having to assume the local availability and in-memory preloading of service registries. The framework also includes an optimal composition search algorithm to extract the best composition from the graph minimising the length and the number of services, and different graph optimisations to improve the scalability of the system, which as far as we now are not included in other frameworks.

2. A reference implementation of this formal framework based on the adaptation of two independently developed components, namely ComposIT [99] and iServe [81], respectively in charge of service composition and discovery.

3. A detailed performance analysis of the integrated system, highlighting both the unacceptable performance achieved when using the typical out of the box discovery implementations, as well as the fact that top performance is achievable with the adequate discovery granularity and corresponding indexing optimisations.

The proposed framework is data-flow centric, focused on the semantic I/O parameter matching of services' interfaces and leaving aside preconditions and effects. This is essentially a pragmatic decision inline with the current tendency towards lightweight data-driven

approaches. In fact, on the Web less than 5% of the semantic Web services include preconditions and effects [82].

The rest of the paper is organized as follows. Sec. 4.3 discusses the state-of-the-art. Sec. 4.4 formalizes the web service composition problem and Sec. 4.5 framework that defines the composition in terms of service discovery tasks. Sec. 4.6 describes our reference implementation. Sec. 4.7 explores the performance of the system for different scenarios and finally Sec. 4.8 gives some final remarks.

4.3 Related Work

Automatic composition of Web services is still an open problem that involves multiple research areas [36]. Concretely, lots of efforts have been devoted to automate the discovery and composition using different approaches and techniques [83]. However, most of the research in both areas has been evolved independently of each other, despite the significant overlap between these interrelated tasks. This has lead to a lack of integrated approaches in the field that consider the performance and the scalability of the overall integrated system as well as the impact of the discovery in terms of response time during the automatic composition task.

From the discovery side, most of the work has been focused on improving the retrieval performance (i.e., precision-recall curve) without much concern about the response time requirements and/or the interface requirements to provide an efficient fine-grained discovery granularity for automatic composition. However, the response time of the discovery systems is recently gaining significant interest. A recent service discovery competition [49] shows some of the newest advances in the automatic discovery field. Most relevant examples are OWLS-MX3 [52], iSem 1.1 [59] and XSSD [30]. The main conclusions that can be drawn from this contest, from the perspective of service composition, are twofold: 1) research efforts are focused on response time improvement via caching and indexing, yet still not sufficient for fast, automatic composition of services and 2) the interface exposed by discovery engines assumes that requests are fully specified in terms of a well-defined interface and categorisation, i.e., discovery systems expect a precise description of the service in terms of inputs and outputs, and/or other characteristics such as preconditions and effects. However, these interfaces are not adequate for service composition, since one of the assumptions is that there is usually no single service that fully matches a request and therefore several services need to be combined instead. Indeed, during automatic composition, an exploratory search is usually

required to guess which relevant services can be selected at each step. This requires to launch many partial requests (fine-grained queries), rather than fully specified requests, in order to locate relevant services that match some partial information available to the algorithm (e.g., services that consume some inputs and/or produce some outputs). Fine-grained requests are simpler and can be solved faster than complex, fully specified requests. Thus they are more suitable for automatic composition systems.

From the composition side, most approaches can be categorized into: 1) classical AI planning approaches [84], where the composition problem is translated into the planning domain and solved using general planners, and 2) graph-based I/O driven approaches that build a graph with the services and their input/output semantic relations (generally ommiting the preconditions and effects), and apply graph search techniques to extract (usually optimal) service compositions from the graph.

Relevant approaches of the first group are [43, 51, 110]. These approaches differ from our work in the sense that they handle very expressive preconditions and effects to generate composition plans but: 1) the concept of external service registries is missing, services are assumed to be locally available; 2) average response time of these systems is usually high; and 3) optimizations to reduce the number of services by identifying redundant functionality are not considered.

On the other hand, graph-based I/O approaches are gaining much attention since the Web Service Challenge [13]. Some notable works in this field are [3, 44, 68, 70, 89, 107, 129]. Concretely, [3, 70, 129] are the top-3 algorithms of the WSC'08. Although these approaches show generally good performance and low response times, [129] and [3] do not find optimal solutions and [70] fails to find solutions in large data sets. Additionally, none of these systems consider neither the integration with service registries nor the use of service optimizations to deal with potential scalability problems.

From the point of view of the integrated frameworks, a very interesting approach was proposed by Kona et al. in [54]. In this paper, the authors present an efficient framework for Web service composition that supports semantic Web service discovery. The composition is generated by performing a forward chaining of operators to find a feasible composition. The authors also evaluated the system with the datasets of the Web Service Challenge 2006 and presented a detailed experimentation. Their results demonstrate the capabilities and the good performance of this system which, however, exhibits some limtations: 1) the notion of an external service registry is missing, all the information required is preprocessed and

loaded in the main memory, which is one of the main issues we set out to tackle with this work since it is otherwise not possible to deal with large and/or distributed datasets; 2) the framework does not contemplate service optimisations to remove redundant information and 3) the work does not perform an optimal search to minimise the cost or the number of services of the composition as all possible compositions with the shortest length are captured in the composition graph which should be further processed to extract the optimal composition. Similarly, in [57], Lécué et al. develop an integrated framework for dynamic Web service composition. The framework exploits the semantic input-output matchmaking to discover relevant services and performs automatic composition using a graph-based approach, taking into account functional and non-functional properties. However, graph optimisations are not considered and the composition search is non-optimal, since the selection of the services is merely greedy-based.

In [34], Da Silva et al. present a framework that effectively supports both automatic semantic discovery and composition, among other relevant phases of the composition life-cycle, such as service publication and service selection, taking into account non-functional properties. One of the limitations of the discovery phase is that it does not support fine-grained requests. On the other hand, the framework does not include neither optimisations to reduce graph size nor an optimal search to extract the best composition from the graph.

In light of the above analysis, we propose a graph-based I/O framework that overcomes all of the analyzed limitations. In this framework the discovery is defined in terms of a fine-grained I/O interface which minimises the performance overhead between both composition and discovery without having to assume the local availability and in-memory preloading of service registries. The proposed framework also includes an optimal composition search algorithm to extract the best composition from the graph minimising the length and the number of services, and different graph optimisations to improve the scalability of the system.

4.4 Web Service Composition Problem

Service composition aims to help construct composite services that could fulfil a user request, e.g., booking an entire holiday, when no known service can achieve such a request on its own. A core activity for creating service compositions is, indeed, the discovery of relevant services. In this context, relevant services are those that could be invoked and contribute to obtaining an executable composition that would fulfil the needs set out by the client. We herein formalise

the composition problem in close relationship with discovery as a means to better study and approach the integration of discovery and composition engines. The formalisation of the problem is data-flow centric, focussed on the semantic input-output parameter matching of services' interfaces.

4.4.1 Semantic Web Service Discovery

The semantic Web service discovery problem consists of locating appropriate services from one or more service registries that are relevant to an input-output request.

Definition 1. *A Semantic Web Service (SWS, hereafter "service") can be defined as a tuple* $w = \{In_w, Out_w\} \in W$ *where* In_w *is a set of inputs required to invoke* w*,* Out_w *is the set of outputs returned by* w *after its execution, and* W *is the set of all services available in the service registry. Each input and output is related to a semantic concept from an ontology* O *(*$In_w, Out_w \subseteq O$*).*

Semantic inputs and outputs can be used to discover relevant services as well as to compose the functionality of multiple services by matching their inputs and outputs together. In order to measure the quality of the match, we need a matchmaking mechanism that exploits the semantic I/O information of the services. The different matchmaking degrees that are typically contemplated in the literature are [76]:

- **Exact** ($\equiv$): An output $o_{w1} \in Out_{w1}$ of a service $w1$ matches an input $i_{w2} \in In_{w2}$ of a service $w2$ with a degree of exact match if both concepts are equivalent.
- **Plugin** ($\sqsubseteq$): An output $o_{w1} \in Out_{w1}$ of a service $w1$ matches an input $i_{w2} \in In_{w2}$ of a service $w2$ with a degree of plugin if o_{w1} is a sub-concept of i_{w2} ($o_{w1} \sqsubseteq i_{w2}$).
- **Subsume** ($\sqsupseteq$): An output $o_{w1} \in Out_{w1}$ of a service $w1$ matches an input $i_{w2} \in In_{w2}$ of a service $w2$ with a degree of subsume if o_{w1} is a super-concept of i_{w2} ($o_{w1} \sqsupseteq i_{w2}$).
- **Fail** ($\perp$): When none of the previous matches are found, then both concepts are incompatible and the match has a degree of fail ($o_{w1} \perp i_{w2}$).

Note that, in order to discover relevant services to generate data-flow compatible service compositions, the only two valid degrees of match are *exact* and *plugin*. On this basis, we define the *cmatch* (compatible match) function that will be used to discover candidate services during the composition phase:

Definition 2. *Given $a, b \in O$, a compatible match cmatch(a,b) holds if and only if $a \equiv b$ (exact match) or $a \sqsubseteq b$ (plug-in match).*

Using the previous compatible match function between concepts, we can define the match-making operator "$\otimes$" that given two sets of concepts $C_1, C_2 \subseteq O$, it returns the concepts from C_2 matched by C_1.

Definition 3. *Given $C_1, C_2 \subseteq O$, we define "$\otimes : O \times O \rightarrow O$" such that $C_1 \otimes C_2 = \{c_2 \in C_2 | cmatch(c_1, c_2), c_1 \in C_1\}$. Note that this operator is not commutative.*

We can use the previous operator to define the concepts of full and partial matching between concepts.

Definition 4. *Given $C_1, C_2 \subseteq O$, a full matching between C_1 and C_2 exists if $C_1 \otimes C_2 = C_2$, whereas a partial matching exists if $C_1 \otimes C_2 \subset C_2$.*

Typically, a service $w = \{In_w, Out_w\}$ is relevant to a request $r = \{In_r, Out_r\}$, where $In_r \subseteq O$ are the provided inputs and $Out_r \subseteq O$ the expected outputs, if $In_r \otimes In_w = In_w$ and $Out_w \otimes Out_r = Out_r$, that is, there is a full match between the provided inputs and the service inputs and a full match between the service outputs and the expected outputs.

While this approach is reasonable for discovering the services that best match an entire request (full match), for composition one needs to locate services that are relevant, that is, that match some inputs / outputs (partial match). Thus, rather than approaching the discovery problem based on a full input/output description, we split this problem into two finer-grained discovery problems that are more relevant for service composition: input discovery and output discovery.

Definition 5. *Given a set of concepts $C \subseteq O$, the input discovery problem can be defined as finding a set of relevant services $W = \{w_1, ..., w_n\}$ where $w_i = \{In_{w_i}, Out_{w_i}\}$ such that $\forall w_i \in W$, $C \otimes In_{w_i} \subseteq In_{w_i}$, that is, services that can consume some (partial match) of the inputs or are directly invokable (full match) with C.*

Definition 6. *Given a set of concepts $C \subseteq O$, the output discovery problem can be defined as finding a set of relevant services $W = \{w_1, ..., w_n\}$ where $w_i = \{In_{w_i}, Out_{w_i}\}$ such that $\forall w_i \in W$, $Out_{w_i} \otimes C \subseteq C$, that is, services that produce some or all outputs.*

Based on these definitions, we introduce the notion of input and output relevance:

Definition 7. *A service $w = \{In_w, Out_w\}$, where $In_w, Out_w \subseteq O$, is input-relevant for a set of concepts $C \subseteq O$ if $C \otimes In_w \neq \emptyset$, whereas the service w, is output-relevant for a set of concepts $C \subseteq O$ if $Out_w \otimes C \neq \emptyset$.*

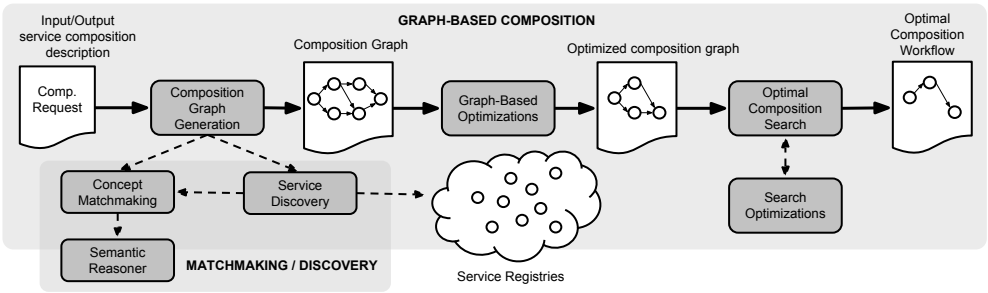

Figure 4.1: Overview of the proposed approach.

4.4.2 Semantic Web Service Composition

The semantic composition problem considered in this work is as follows: Given a request $r = \{In_r, Out_r\}$, where In_r is a set of available semantic input concepts and Out_r a set of requested semantic output concepts, we can define the problem of the automatic construction of a SWS composition as that of finding a composite Web service $w_c = \{In_{w_c}, Out_{w_c}, P = \{S, \leq\}\}$ such that $In_r \otimes In_{w_c} = In_{w_c}$ (the composite service is invokable with the available inputs) and $Out_{w_c} \otimes Out_r = Out_r$ (the composite service retrieves all the requested outputs). This service consists of a partially ordered set P (a binary relation "$\leq$" over a set of services $S \subseteq W$). This partial ordered set of services is esentially a Directed Acyclic Graph (DAG) which models the implicit execution order of the services driven by the input/output matches, where nodes of the DAG are services and the arcs are valid semantic matches. This type of composition has many advantages: On one hand, mapping inputs and outputs to semantic concepts does allow to reason about data types to improve the matchmaking between service parameters, which leads to more possible semantically valid compositions. On the other hand, DAG representation formally captures the nature of a composition where services may be executed in different orders, i.e., there are many different total (sequential) orderings of a composition that lead to the same result. Moreover, since our approach is data-flow centric, a

DAG representation is simpler than a general (possible cyclic) graph as cycles do not produce new data types in the composition.

However there are also some drawbacks. First, a DAG representation could impose some restrictions in the compositions that can be generated, i.e, due the absence of cycles, a service could not explicitly be invoked twice. Second, compositions at different semantic levels rather than just concept matchmaking would deffinitely improve the quality of the compositions by capturing more possible cases. Furthermore, using input concepts and output concepts to define a composition request is not user friendly. A better way to specify a request would be to define it with keywords. This, nonetheless, could be achieved with a pre-processing step using automatic semantic annotation tools to translate the request from keywords to semantic concepts. Formally, we define a valid composition as follows:

Definition 8. *Let $r = \{In_r, Out_r\}$ and let $w_c = \{In_{w_c}, Out_{w_c}, P = \{S, \leq\}\}$ be a composite service for the request r, where P is a partial order over the set of services $S \subseteq W$ of the composite service w_c. We say w_c is a valid composition for request r if and only if, for any topological sort $T = \{w_1, w_2, ..., w_N\}$ of P, where $w_j = \{In_{w_j}, Out_{w_j}\}$ $\forall j \in [1, N]$, the following expression is satisfied:*

$$(In_r \otimes In_{w_1} = In_{w_1}) \wedge ((In_r \cup Out_{w_1}) \otimes In_{w_2} = In_{w_2})$$
$$\wedge \ldots \wedge ((In_r \cup Out_{w_1} \cup \ldots \cup Out_{w_N}) \otimes Out_r = Out_r).$$

This definition implies that every service of the composition must be invokable to obtain an invokable service composition. We say that a service $w = \{In_w, Out_w\}$ is invokable with a set of concepts $C \subseteq O$ if each required input $i_w \in In_w$ is semantically matched by a set of concepts C.

Definition 9. *If $C \subseteq O$ is the set of available input concepts, then a service $w = \{In_w, Out_w\}$ is invokable with C if $C \otimes In_w = In_w$, i.e., there exists a full matching between the available inputs and the service inputs.*

Note that if a service w is invokable with a set of concepts C, then it is also input-relevant for the same set of concepts since *invokable* implies *input-relevant*, but the inverse does not hold (see Def. 7). That is, the set of invokable services is included in the set of the relevant services.

The reader should note that we restrict the definition of a compatible match to *exact* and *plugin* in order to generate semantically complete compositions. However, the framework also

supports the use of other match degrees (e.g., subsume) by relaxing the "$cmatch$" operator, which in practice means obtaining potentially more matched (but semantically weaker) concepts and thus bigger composition graphs with more services and match relations that could be semantically incomplete. This is supported not only in theory, but also by the reference implementation presented in Sec. 4.6.

4.5 Composition Framework

On the basis of the formal definition of the problem, in this section we present a graph-based framework for automatic semantic Web service composition. Fig. 4.1 shows the overview of our approach with the different steps involved. The process is triggered by a composition request that specifies the user requirements in terms of inputs and the expected outputs. This information is used in the composition graph generation phase to build a graph with all the relevant services and the semantic relations between their inputs and outputs. In order to find the relevant services, the composition graph phase is interleaved with the *discovery phase*. The discovery phase is responsible for retrieving the relevant services given the data available at different stages during the *composition graph generation* phase. The relationships between the inputs and outputs of services are computed in the *matchmaking phase*, where the semantic matching degree between inputs and outputs is computed using a semantic reasoner. The service composition graph is eventually generated on the basis of the relevant services and the I/O matching information. This graph contains all possible service compositions that satisfy the composition request, in addition to a few others that, although invokable, do not manage to entirely fulfil the request. The service composition graph is then optimised applying different techniques to group and reduce the number of services and relations. Next, an *optimal search* is performed over the graph to find the optimal composition. This phase is interleaved with a *search optimisation phase* that analyses and reduces the search space. Finally, the optimised composition workflow is returned.

In this section, we analyse each phase and we provide generic strategies based on the problem description presented in the previous section.

4.5.1 Semantic Matchmaking

A fundamental functionality that needs to be available for generating compositions and even for discovering services, is the ability to analyse the compatibility between different seman-

tic types. This functionality, which we refer to as semantic matchmaking, is in charge of assessing the level of semantic compatibility between concepts, given an ontology (or set of ontologies). To do so, semantic matchmaking relies on semantic reasoning (notably subsumption reasoning) in order to be able to determine the relationships between the concepts (e.g., Plugin match). This mechanism can be used for example, to discover services that can consume or produce a concrete input/output by finding semantically compatible types. Such a mechanism is also particularly relevant for generating the service composition graph with all the matches between services inputs and outputs.

The matchmaking system provides a $match(C_1, C_2)$ function which represents the concrete implementation function of the $\otimes$ operator defined in Def. 3. The *match* function tries to find a valid match between the *source concepts* of C_1 and the *target concepts* of C_2 calling the $cmatch(c_i, c_j)$ function (Def. 2) for each pair (c_i, c_j) of concepts where $c_i \in C_1$ and $c_j \in C_2$. The compatible match function is calculated using a semantic reasoner that returns the semantic relation between two concepts. Then, it checks if the relation is considered a compatible match (i.e., *exact* or *plugin*). Each time a compatible match is found between c_i and c_j, c_j is added to a set of matched concepts and removed from C_2. The reader should note that the goal here is not to find the best match for each element but rather to get all compatible matches for each target element.

The best-case complexity (all C_2 concepts matched by the first element from C_1) is $O(m)$, whereas the worst-case complexity (no compatible matches at all) is $O(m \cdot n)$ where $n = |C_1|, m = |C_2|$. This implies that, in the worst case, for two sets of elements, there will be at most $m \times n$ calls to the *cmatch* function which is ultimately answered by the semantic reasoner.

4.5.2 Semantic Service Discovery

In order to generate service compositions, it is necessary to be able to discover appropriate services based on their interface. The goal of a typical discovery system is to find atomic services that match entirely a description representing the ideal service sought, i.e., all the inputs and outputs are compatible. However, from the viewpoint of generating data-flow compatible compositions, rather than looking for entire matches, we need to find suitable combinations of services that combined would satisfy a request. In this scenario, the ability to find partially matching services very fast is paramount in order to enable exploring efficiently the many possible combinations of services that could lead to a suitable composition. Therefore, in a nutshell, the type of service discovery that is required for supporting service composition is a

more relaxed and finer-grain version of that typically provided by discovery engines whereby partial matches can be obtained in a very fast manner. This can be achieved by defining a simple fine-grained interface that supports the discovery of services using only partial information (some/any available inputs, some/any expected outputs). Fig. 2 shows the pseudocode of this simple interface to discover relevant services that can be used as a starting point to obtain semantic input/output relevant services, as defined in Def. 7 in Sec. 4.4.

The discovery algorithm sequentially scans all services and calls the $Match$ function of the *Matchmaker* to determine if a service is relevant for an input (the service has at least one input compatible with the inputs provided) or for an output (the service has at least one output compatible with the outputs provided) depending on the $Type$ selected. Therefore, the complexity of this type of discovery is $O(w)$ where $w = |W|$ is the size of the service repository. This implies at most $|W|$ calls to $Match$ in the worst-case scenario or $O(w \cdot m \cdot n)$ if we consider the complexity of the $Match$ method assuming every service has at most m outputs and n inputs.

Algorithm 2 Pseudocode to obtain input-relevant and output-relevant sets of services for a particular set of concepts

```
1:  function RELEVANTIO(C ⊆ O, W, type)
2:      relevantServ := {}
3:      for all w_i = {I_wi, O_wi} ∈ W do
4:          if type = In then
5:              if match(C, I_wi) then
6:                  relevantServ := relevantServ ∪ w_i
7:          else if type = Out then
8:              if match(O_wi, C) then
9:                  relevantServ := relevantServ ∪ w_i
10:     return relevantServ
```

4.5.3 Service Composition Graph Generation

When the system receives a request, the *Service Composition Graph Generator* computes a graph with all the semantic relations between the relevant services for the request. A request is basically a set of input concepts, which represent the initial set of available inputs, and a set of output concepts, which are the outputs that the composite service should return. The *service composition graph* is basically a layered Directed Acyclic Graph (DAG), $G = (V, E)$,

where:

- $V = W \cup C$ is the set of vertices of the graph, where W is the set of services and C the set of concepts (inputs and outputs).
- $E = CW \cup WC \cup CC$ is the set of edges in the graph where:
  - $CW \subseteq \{(c,w) \mid c,w \in V \wedge c \in C \wedge w \in W\}$ is the set of input edges, i.e., edges connecting input concepts to their services.
  - $WC \subseteq \{(w,c) \mid w,c \in V \wedge w \in W \wedge c \in C\}$ is the set of output edges, i.e., edges connecting services with their output concepts.
  - $CC \subseteq \{(c,c') \mid c,c' \in V \wedge c,c' \in C \wedge cmatch(c,c')\}$ is the set of edges that represent a semantic match between concepts.

This graph contains all the known services that could directly or indirectly be invoked given the provided inputs. The graph is divided into N layers, whereby each layer i has all those services whose inputs are matched by the outputs produced in previous layers and, therefore, are invokable at layer i. The graph is augmented with two layers, namely L_0 and L_{N+1}. L_0 contains the dummy service $w_O = \{O_R, \emptyset\}$ whereas L_{N+1} contains the dummy service $w_I = \{\emptyset, I_R\}$. The first one is a service that provides as outputs the inputs of the request (I_R) and the last one has the goal outputs (O_R) as inputs. An example of a graph for I_R={*BookTitle, BookAuthor, CreditCard, Email, Address*} and O_R={*Price, Payment, BookingCode*} is shown in Fig. 4.2.

The first step of the composition graph construction is the calculation of the relevant services. These services can be easily calculated forwards, layer by layer, using the discovery mechanism previously presented. Fig. 3 shows an implementation of the forward composition graph generation algorithm for a request R. The algorithm selects all those services from the set of all available services W that are input-relevant for the available concepts (*availCon*) in each layer using the *relevantIO* function (L. 8). Then, for each *input-relevant* service, the algorithm performs a match between the available concepts and the unmatched inputs of each service. All the inputs that are matched are removed from the unmatched set of inputs for the current service. If there are no unmatched inputs, then the service is invokable and thus is eligible for the current layer. For example, the first eligible services for the request shown in Fig. 4.2 are the services in the layer $L1$, which correspond with the services whose inputs are fully matched by I_R (the set of concepts in $L0$). The second eligible services are those

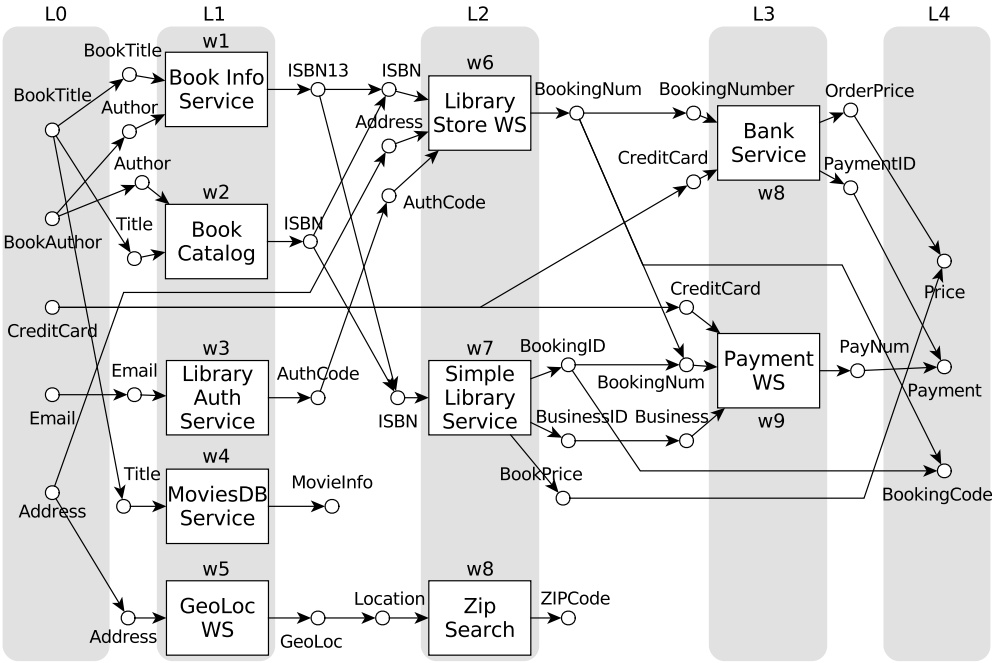

Figure 4.2: Composition graph example.

services (placed in $L2$) whose inputs are fully matched by the outputs of the previous layers, and so on. Note that instead of performing the invokability check by finding a full match between C and the inputs of each service, we save those inputs of each service that have been matched before, and hence we only perform the match between the new outputs generated in the previous level (*availCon*) and the remaining unmatched inputs of each service (U_{set}). Hence, the unmatched inputs U_{set} of each service decreases monotonically with each level (i.e., the unmatched inputs of each service always decrease when a new match is found, and the effect is propagated at each layer). The complexity analysis for this algorithm (neglecting the optimisation effect due to the propagation of the matched inputs for simplification purposes) is $O(l \cdot w \cdot m \cdot n + l \cdot \frac{w}{k} \cdot m \cdot n)$ which can be simplified to $O(l \cdot m \cdot n(\frac{(k+1)w}{k})$. The first part corresponds with the complexity of the calls to the *relevantIO* function which is invoked l times (one call per layer), whereas the second part corresponds with the complexity of the *for* loop to check the invokability of each *input-relevant* service. We can expect that only a

small subset of the repository W is relevant for the *availCon* generated in the previous layer. Thus, each call to *relevantIO* function returns a small set of relevant services w/k where k ($k \gg 1$) is a reduction factor that depends on the number of relevant services for a given set of concepts. This k factor is different for each request and service registry. For example, if we assume $k = 100$ for a given problem for a service registry of 1,000 services, then it means that each invokation of $relevantIO(availCon, W, In)$ will return only the 1% of the services of the repository ($w/k = 10$). Consider the following example of a composition over a repository with 1,000 services ($w = 1,000$), assuming that there are $m = 5$ new output concepts generated and $n = 5$ unmatched concepts at each layer, the composition graph has 10 layers ($l = 10$) and in each layer the *relevantIO* function returns on average $w/k = 10$ services (that is, $k = 100$). The complexity in this example is $10 \cdot 1000 \cdot 5 \cdot 5$ for the first part plus $10 \cdot \frac{1000}{100} \cdot 5 \cdot 5$ for the second part, which is $\approx 2.5 \cdot 10^5$ calls to the matchmaking system to compute all the required matches at the concept level.

Algorithm 3 Algorithm for forward graph generation.

1: **function** FWDGRAPH($R = \{I_R, O_R\}, W$)
2: $C := I_R$; $i := 0$; $L_0 := \{w_I\}$; $L := L_0$
3: $unmatchedIn := [\,]$; $availCon := I_R$
4: $W' := W$;
5: **repeat**
6: $i := i + 1$
7: $L_i := \emptyset$; $W_{selected} = \emptyset$
8: $W_{relevant} := relevantIO(availCon, W', In)$
9: $availCon := \emptyset$
10: **for all** $w_i = \{I_{w_i}, O_{w_i}\} \in W_{relevant}$ **do**
11: $U_{set} := unmatchedIn[w_i]$
12: $M_{set} := Match(availCon, U_{set})$
13: $unmatchedIn[w_i] := U_{set} \setminus M_{set}$
14: **if** $M_{set} = \emptyset \wedge w_i \notin L$ **then**
15: $W_{selected} = W_{selected} \cup w_i$
16: $availCon := availCon \cup O_{w_i}$
17: $L_i := L_i \cup W_{selected}$
18: $W' := W' \setminus W_{selected}$
19: $C := C \cup availCon$
20: **until** $(Match(C, O_R) = O_R) \vee L_i = \emptyset$
21: $L := L \cup \{w_O\}$

Index-Based Optimisations

Although these improvements can save search time, one of the bottlenecks of the graph generation is still the size of the repository w, which is usually some orders of magnitude bigger than the other parameters involved in the complexity. One effective way to reduce the impact of the size of the repository is precalculating and indexing the input-relevant set of services for each concept of the ontology. The indexing of services can be done independently of any composition request as it only depends on the information available, such as the services themselves and the ontologies.

The construction of an inverted index function to recover input-relevant services or output-relevant services can be done easily using the *relevantIO* function. The main idea behind the inverted index is to build a key-value hash map where the keys are the concepts of the ontology and the values are those services that are input-relevant (or output-relevant) for that concept. This map allows to discovery relevant services in constant time during the graph generation.

We define a new function $relevantIO'$ which is the cached-version of the original function. Instead of computing the relevance by using directly the matchmaking system, it first checks if the concept is cached in the inverted index. If the concept is in the index, then it is immediately returned (constant time). If not, the call is delegated to the *relevantIO* function. Assuming there is enough memory to keep the entire index, the index allows to provide relevant services at $O(1)$ for each concept during the forward graph generation. Thus, we reduce the complexity associated to the parameter w. Concretely, since we can obtain at constant time the input-relevant services for each concept, the complexity of $relevantIO(availCon, W, In)$ now depends only on the number of concepts in *availCon* (one access to the index per concept). Having $m = |availCon|$ (number of new concepts at each layer) the complexity using indexes is $O(l \cdot m + l \cdot \frac{w}{k} \cdot m \cdot n)$, simplified to $O(l \cdot m(1 + \frac{w}{k} \cdot n))$. The use of indexes to discover relevant services during the forward graph generation has a high impact on the global performance. Using the same example as before, with $w = 1000$, $l = 10$, $m = 5$, $n = 5$ and $k = 100$ we have $10 \cdot 5(1 + \frac{1000}{100} \cdot 5) = 2.55 \cdot 10^3$, 2 orders of magnitude lower than the non-indexed version.

4.5.4 Graph-Based Optimisations

Once the graph is generated, the next step is to apply different optimisations to reduce the graph size in order to improve the optimal composition search performance. This part of the

composition is independent of the discovery phase. All the information required to search for the optimal composition is in the graph, namely, the relevant services and the semantic relations between their inputs and outputs, so there is no need to communicate with the discovery/matchmaking systems. We distinguish at least two different techniques [96, 99]: *backward pruning* and *interface dominance*.

Backward pruning

As explained earlier, the generation of the composition graph with the relevant services is done forwards, layer by layer. During this forward expansion of the graph, we are not interested in invoking services that have no explicit effects on the composition, that is, services that are not contributing to the output goals. When the graph is completed and the goal outputs are reached, a backward pruning is performed to remove all non-contributing services. A non-contributing service is essentially a service that is not contained in the *transitive closure set* of the *output-relevant* services. A service $w' = \{In_{w'}, Out_{w'}\}$ is *output-relevant* for a service $w = \{In_w, Out_w\}$ if $Out_{w'} \otimes In_w \neq \emptyset$ (def. 7). Thus, the set of all *output-relevant* services for a service w can be defined as:

$$X(w) = \{w' \in W \mid Out_{w'} \otimes In_w \neq \emptyset\} \tag{4.1}$$

Recursively, we can define the set of $X^2(w) = X(X(w))$ as the set of *output-relevant* services at the distance two. Extending this, the *transitive closure* of the *output-relevant services* can be defined as:

$$\hat{X}(w) = X(w) \cup X^2(w) \cup X^3(w) \cup \cdots \tag{4.2}$$

Therefore, we can say that all those services of the graph that are not in the *transitive closure* of the *output-relevant* services $\hat{X}$ are not contributing to the composition goals, directly nor indirectly, and can therefore be removed from the graph.

An example of this can be seen in Fig. 4.2. Starting from the last layer, we compute the transitive closure of the service w_O, which is a dummy service that represents the goal outputs. The output relevant services for w_O at distance one are $X(w_O) = \{w_6, w_7, w_8, w_9\}$, since $Out_{w_6} \otimes In_{w_O} \neq \emptyset$ and the same for w_7, w_8 and w_9. We calculate now the output-relevant services at distance two, which is $X(X(w_O)) = X(\{w_6, w_7, w_8, w_9\})$. $X(\{w_6, w_7, w_8, w_9\})$ can be simply computed as the union of $X(w_6) \cup X(w_7) \cup X(w_8) \cup X(w_9)$ which is $\{w_1, w_2, w_3\}$.

Repeating this, we finally have $\hat{X} = \{w_6, w_7, w_8, w_9\} \cup \{w_1, w_2, w_3\} \cup \{w_I\}$, where w_I is the dummy service ommited in Fig. 4.2 that provides the input concepts of the request (concepts in L_0). Since $w_4, w_5, w_8 \notin \hat{X}$, these services ($w_4$=*MoviesDB Service*, w_5=*GeoLoc WS*, w_8=*Zip Search*) are not contributing to the goals and can be removed from the graph.

Interface Dominance

Another strategy to reduce the graph size is to analyse the equivalence and dominance of some services over others in terms of the interface they offer. It is very frequent to find services from different providers that offer similar services with overlapping interfaces. In scenarios like this, it is easy to end up with large composition graphs that make very hard to find optimal compositions in reasonable time. One way to attack this problem is to analyse the *interface dominance* between services in order to find those that are equivalent or better than others in terms of the interface they provide.

Definition 10. *Given a concept in a composition graph G ($c \in G$), we denote $\Phi(c)$ as a function that returns the set of output-relevant services for concept c:*

$$\Phi(c) = \{w = \{In_w, Out_w\} \in G \mid Out_w \otimes \{c\} = \{c\}\} \tag{4.3}$$

For instance, $\Phi(Payment)$ in Fig. 4.2 is $\{w_8, w_9\}$ since $Out_{w_8} \otimes \{Payment\} = \{Payment\}$ and $Out_{w_9} \otimes \{Payment\} = \{Payment\}$, that is, concept *Payment* is matched by an output from w_8 (*PaymentID*) and for an output from w_9 (*PayNum*).

Definition 11. *A service $w_i = \{In_{w_i}, Out_{w_i}\} \in G$ is input-equivalent ($In_{w_i} \equiv In_{w_j}$) with respect to a service $w_j = \{In_{w_j}, Out_{w_j}\} \in G$ in the composition graph G if:*

$$\bigcup_{c_i \in In_{w_i}} \{\Phi(c_i)\} = \bigcup_{c_j \in In_{w_j}} \{\Phi(c_j)\} \tag{4.4}$$

That is, the set of sets defined by the union of $\Phi(c)$ for each input concept c of each service must be equal. This definition formalises the idea of input equivalence of two services of the composition graph regarding the relation between their inputs and the services that match those inputs. That means that two services w_i and w_j of the graph are input equivalent if the services that provide the inputs of both services are the same.

Definition 12. *A service $w_i = \{In_{w_i}, Out_{w_i}\} \in G$ is input-dominant ($In_{w_i} \succ In_{w_j}$) with respect to a service $w_j = \{In_{w_j}, Out_{w_j}\} \in G$ in the composition graph G if:*

$$\bigcup_{c_i \in In_{w_i}} \{\Phi(c_i)\} \subset \bigcup_{c_j \in In_{w_j}} \{\Phi(c_j)\} \tag{4.5}$$

Thus, informally, a service is input-dominant if it only needs a subset of the information required by the dominated service to be invoked. For example, in Fig. 4.2, w_7 is input-dominant respect to w_6, since $\{\{w_1, w_2\}\} \subset \{\{w_1, w_2\}, \{w_I\}, \{w_3\}\}$.

Definition 13. *Given a concept in a composition graph G ($c \in G$), we denote $\Psi(c)$ as the function that returns a set of input concepts in G that are matched by c, that is, there exists an arc from c to c' in G.*

$$\Psi(c) = \{c' \mid (c, c') \in G\} \tag{4.6}$$

Definition 14. *A service $w_i = \{In_{w_i}, Out_{w_i}\} \in G$ is output-equivalent ($Out_{w_i} \equiv Out_{w_j}$) respect to a service $w_j = \{In_{w_j}, Out_{w_j}\} \in G$ in the composition graph G if:*

$$\bigcup_{c_i \in Out_{w_i}} \Psi(c_i) = \bigcup_{c_j \in Out_{w_j}} \Psi(c_j) \tag{4.7}$$

That is, two services are output-equivalent if their outputs are matched to the same input concepts in the graph, which means that their outputs can be consumed in the same way by the same services in G.

Definition 15. *A service $w_i = \{In_{w_i}, Out_{w_i}\} \in G$ is output-dominant ($Out_{w_i} \succ Out_{w_j}$) respect to a service $w_j = \{In_{w_j}, Out_{w_j}\} \in G$ if:*

$$\bigcup_{c_i \in Out_{w_i}} \Psi(c_i) \supset \bigcup_{c_j \in Out_{w_j}} \Psi(c_j) \tag{4.8}$$

Therefore, one service is *output-dominant* with respect to another service of the graph G if their outputs match the same inputs of the same services in the composition graph but the dominant service also provides additional outputs to the same or different services.

Definition 16. *a service $w_i = \{In_{w_i}, Out_{w_i}\}$ is interface-equivalent to a service $w_j = \{In_{w_j}, Out_{w_j}\}$ ($w_i \equiv w_j$) if $In_{w_i} \equiv In_{w_j}$ and $Out_{w_i} \equiv Out_{w_j}$, that is, both are input-equivalent and output-equivalent.*

Definition 17. *A service w_i interface-dominates a service w_j ($w_i \succeq w_j$) if the first dominates the second in at least one aspect (input-dominant or output-dominant) and is at least equivalent in the other aspect. Formally, $w_i \succeq w_j$ if $(In_{w_i} \succ In_{w_j} \wedge Out_{w_i} \succ Out_{w_j}) \vee (In_{w_i} \equiv In_{w_j} \wedge Out_{w_i} \succ Out_{w_j}) \vee (In_{w_i} \succ In_{w_j} \wedge Out_{w_i} \equiv Out_{w_j})$.*

This dominance definition can be generalised to include more features, such as preconditions, effects, or non-functional properties like QoS:

Definition 18. *A service with multiple properties $w_i = \{P^1_{w_i}, P^2_{w_i}, \ldots, P^n_{w_i}\}$ where $P^1_{w_i}$ are the inputs, $P^2_{w_i}$ the outputs and the rest of parameters are different properties, dominates another service w_j ($w_i \succeq w_j$) with parameters $P_{w_j} = \{P^1_{w_j}, P^2_{w_j}, \ldots, P^n_{w_j}\}$, if $\forall\, k \in \{1,...,n\}$ $P^k_{w_i} \succeq P^k_{w_j} \wedge \exists\, k \in \{1,...,n\}, P^k_{w_i} \succ P^k_{w_j}$.*

The interface dominance optimisation allows to reduce the size of the composition graph by substituting the original services of the graph by abstract interfaces that capture the functionality of the dominant or equivalent services. By minimising the graph size we improve the performance of the search algorithms since they only explore a reduced search space. Once the search is performed and the optimal composition workflow is generated, a post-processing step can be used to replace the abstract service interfaces with specific implementations using the original dominant / equivalent services or by combinations of dominated services that satisfy the same functionality of the dominant service.

4.5.5 Optimal Composition Search

The previous optimisations are intended to reduce the composition graph but keeping the same functionality. The next step is to perform a search over the graph to find the best composition among all the possible compositions that satisfy the input/output request. The search can be designed to optimise different criteria, such as the number of services, the execution path length or QoS properties. Typically, the search over the graph can be done forwards or backwards. In the first case, the composition starts from the inputs of the request (first layer), selecting invokable services until the goal outputs are obtained, whereas the second case starts with the goal outputs (last layer), selecting relevant services for the outputs until a composition that can be invoked with the initial inputs is found.

Formally, the composition search can be modelled as a state-transition system, where the problem is divided into a set of states and transitions between states [39]. A state transition system is defined as a 3-tuple $\Sigma = (S, A, \gamma)$, where:

- $S = \{s_1, s_2, \dots\}$ is a finite set of states.
- $A = \{a_1, a_2, \dots\}$ is a finite set of actions.
- $\gamma : S \times A \to S$ is a state-transition function.

Using the concept of the state-transition system, the state space search problem can be defined as $P = \{\Sigma, s_0, G\}$, where $s_0 \in S$ is the initial state and $G \subseteq S$ is a set of goal states.

The state-transition system Σ allows the search to navigate through the set of states applying different actions, where each action may be associated to a cost that we want to minimise. The state representation may vary depending on the strategy used. Typically, in the case of the backward search, the state will contain the information of the unsatisfied concepts at each state, starting with the goal outputs. The goal then is to find a succession of actions $\langle a_1, a_2, \dots, a_n \rangle$ with the minimum cost that leads from the initial state, where unsatisfied concepts = goal outputs, to the goal state, where unsatisfied concepts = $\emptyset$, that is, there are no unsatisfied concepts and the composition is invokable. The available transitions between states are given by the applicable actions to each state, i.e., the output relevant services that can be selected to resolve all the unsatisfied concepts.

Given a composition graph $G = (V, E)$ as defined previously, where $V = W \cup C$ is the set of vertices which are the services and the concepts (inputs/outputs) of the graph, the state-transition system Σ for the (backward) composition problem is defined as follows:

- $S \subseteq 2^C$ where C is the set of all concepts in the composition graph, i.e., a state is a set of concepts of the graph, $s = \{c_1, \dots, c_n\}$.
- $A \subseteq 2^W$ where W is the set of services in the composition graph, i.e., an action is a set of services from the graph, $a = \{w_1, \dots, w_n\}$.
- $\gamma(a, s) = (s - \bigcup(\Psi(c_i) \mid c_i \in Out(a)) \cup In(a))$, i.e., the application of an action $a = \{w_1, \dots, w_n\}$ to a state $s = \{c_1, \dots, c_n\}$ generates a new state where all concepts that are matched by the outputs of the services of the actions are removed, and the inputs of the services of the actions are added as the new unsatisfied concepts. Functions $In(a)$ and $Out(a)$ return the union of the input concepts and the union of the output concepts of the services in a respectively.

The initial state s_0 of the backward composition problem $P = (\Sigma, s_0, G)$ is defined as $s_0 = In_{w_O}$, i.e., the input concepts of the output dummy service. For example, in Fig. 4.2, the initial

state is $s_0 = \{i_{18}, i_{19}\}$. On the other hand, there is just one goal state $G = \{s_g = \emptyset\}$, i.e., the goal state is reached when there are no unsatisfied concepts in the composition.

The efficiency of the search can also be improved using *search optimisations* depending on the search strategy followed. These optimisations can be applied to the available actions for each state by pruning actions that lead to dead-ends, actions that are equivalent, or actions that are dominated (cannot lead to a better solution).

4.6 Reference Implementation

We developed a reference implementation of the integrated graph-based composition framework that is based on two main components: iServe [81], a service warehouse with advanced discovery support which provides the service registry and takes care of the matchmaking and service discovery activities, and ComposIT [99], which is in charge of the graph-based composition part.

Fig. 4.3 depicts the architecture of the system. In a nutshell the composition process is carried out as follows. When a composition request is sent to the system through the Web UI, ComposIT starts computing the composition graph with all the relevant services for the request. To this end, all the relevant services are discovered layer by layer using the fine-grained I/O logic-based discovery support provided by the Semantic Discovery Engine of iServe. This engine relies on the *Service Manager* and the *KB Manager* to retrieve the relevant services using semantic reasoning capabilities. During the composition graph generation, ComposIT also makes intensive use of the *KB Manager* in order to carry out concept level matching and consequently figure out how the inputs and outputs of the services obtained can be connected. Once the composition graph is generated, ComposIT applies the *backward pruning* and the *interface dominance* optimisations to reduce the graph size. These optimisations are applicable using only the information contained in the graph, and thus there is no need to interact with the discovery component. Finally, an optimal search is performed over the graph using a backward A* algorithm that extracts the optimal composition from the graph.

In the next sections we shall cover in more detail the inner workings of iServe and ComposIT respectively.

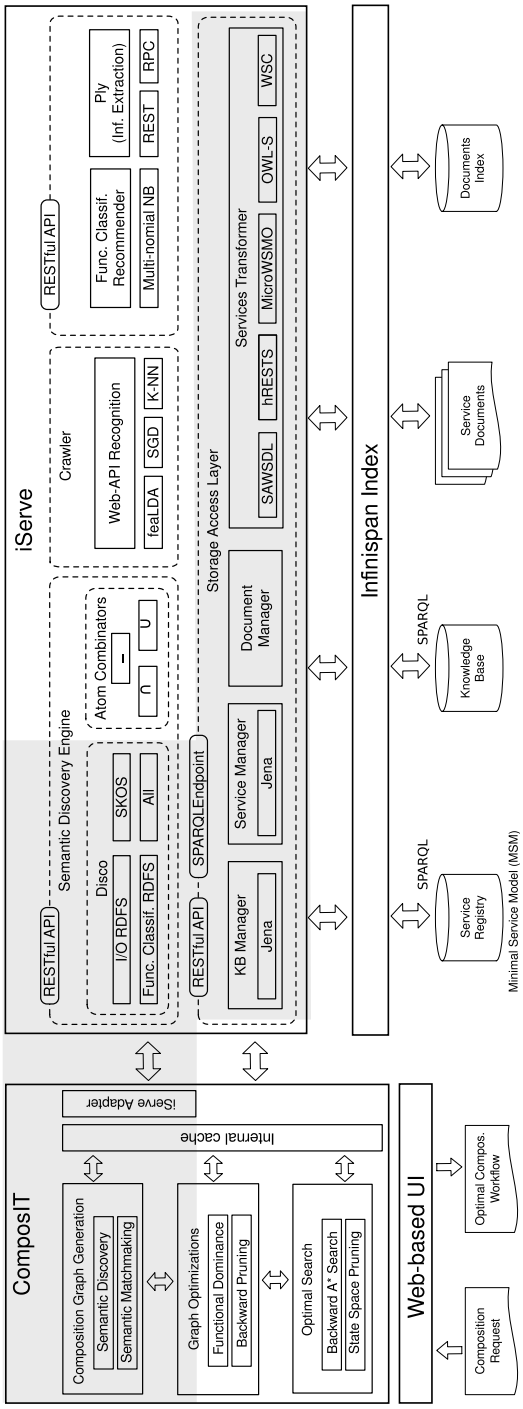

Figure 4.3: ComposIT / iServe architecture

4.6.1 iServe

iServe [81], see right hand-side of Fig. 4.3, is a service warehouse whose functionality includes the core service registry anchored on Linked Data principles, semantic reasoning support, advanced discovery functionality, and further analysis components able to assist in automatically locating and generating semantic service descriptions out of Web resources. For the purposes of this work we have essentially exploited the registry and discovery functionality.

The service discovery functionality builds on top of the Storage Access Layer, which is in charge of managing the registry's data that includes Service descriptions, related documents and the corresponding Ontologies. This layer essentially provides a RDF/S and OWL storage and reasoning support, document storage, as well as basic crawling facilities to automatically obtain referenced Ontologies. RDF/S and OWL storage and reasoning support is delegated to dedicated engines which are accessed by means of the SPARQL 1.1 standard. Therefore, the reasoning capabilities depend largely on the actual configuration of the store. Concretely, the discovery infrastructure contacts the Service Manager to list services given basic criteria such as the input and output types provided, and the KB Manager to obtain concepts, properties, and their sub or super concepts. Depending on their implementation Service and KB Managers combine internal indexes with *SPARQL* queries issued to the triple store by means of Jena.

Services are imported to iServe using a range of transformation engines able to import service descriptions in a variety of formalisms including SAWSDL, WSMO-Lite, OWL-S, and MicroWSMO. These plugins generate descriptions expressed in terms of a simple RDF/S model, Minimal Service Model (MSM) [81], which essentially captures the intersection of existing service description formalisms. By means of these transformations iServe provides an homogeneous description for services that were orginally annotated using heterogeneous means.

Given that, as we saw in Section 4.5, the response time of the overall composition is highly dependent on the performance of the service discovery and concept matchmaking tasks, we extended iServe with various implementations of the Service and Knowledge Base Managers. We tested different configurations to study their individual performance and the overall impact on composition response times. In particular, we used the following configurations:

1. *SPARQL D/M*: pure *SPARQL* Discovery / Matchmaking where all interactions with the Service and Knowledge Base managers are directly implemented as *SPARQL* queries.

This is the typical approach of discovery engines and was the original implementation of iServe.

2. *Index. D/SPARQL+Cache M*: I/O service discovery is based on an index. We additionally used herein an intermediate cache at the level of the concept matcher in order to avoid issuing recurrent SPARQL queries.

3. *Full Indexed D/M*: both service discovery and concept matchmaking relied on local indexes pre-populated at load time (and updated with writes). In this configuration, service discovery and concept matchmaking do not need to issue any SPARQL query to the backed.

4.6.2 ComposIT

ComposIT [99], depicted in the left hand-side of Fig. 4.3, is the semantic Web service composition engine we rely on. It implements all the different graph-based composition phases of the framework described in Sec. 4.5. The semantic service discovery and matchmaking mechanisms, which originally were directly implemented internally, are delegated to iServe by means of integration adapters implemented for the purposes of this work. ComposIT nonetheless uses an internal cache and an index to efficiently recover the information of the generated composition graph. It is worth to note that the architecture supports the deployment of multiple, distributed iServe instances to provide different endpoints that can be used by ComposIT in the composition phase by aggregating the results of the registries at the ComposIT API level. Indeed, since the services to contemplate at composition time are identified by the remote registry and we just use them directly, composing this set of services out of just one API call or several calls in parallel (one per registry) is a trivial change. The overall response time analysis would still remain unchanged, and would have an upper-bound determined by the slowest registry. This also applies to other third-party discovery engines as long as they support fine-grained I/O discovery queries as described in Sec. 4.5.2. The integration of these third-party registries could be achieved by developing interface adapters (with capabilities to retrieve input and output relevant services) which could be plugged in to the system, keeping the generation of the composition graph isolated from the concrete registries used.

The generated composition graph can contain different compositions with the same or different length (number of layers) and with different number of services depending on the services that have been selected to generate the needed data. Among the different combina-

tions that can be obtained, the goal of ComposIT is to find the shortest service composition with the minimum number of services. For this purpose, ComposIT searches for the optimal composition by carrying out a heuristic search based on the A* algorithm [40]. This search was implemented using Hipster4j [95] to identify a minimal subset of the services from the graph that satisfy the request (in terms of inputs and outputs). Note that multiple compositions can be extracted from the composition graph since there may be different services that generate outputs of the same concept.

4.7 Evaluation

In this section we present a quantitative evaluation of our approach. The purposes of the evaluation are: 1) measure the scalability of the approach with many services; 2) study the impact of the discovery on the overall composition performance and 3) compare the performance with different optimisations.

In order to perform a standard and comparable evaluation, we selected the Web Service Challenge 2008 (WSC'08) service datasets. These datasets allow us to measure the scalability with an increasingly large set of services (from 158 to 8,119 services). Services were imported to iServe using an specific transformer plugin which translates each service description in the WSC'08 XML format into MSM, and the XML concept taxonomy into an equivalent OWL representation. iServe is responsible of identifying, loading and reasoning with the ontologies used in the service descriptions. Data types of the input and outputs of service descriptions are linked to their corresponding semantic concepts through the $modelReference$ property of the MSM, which points to the concepts defined in the transformed OWL model.

Experiments were run under Ubuntu 10.04 64-bit on a PC with an Intel Core 2 Duo E6550 at 2.33GHz and 4 GB of RAM. OWLIM-Lite 5.3 with OWL Horst reasoning was chosen in iServe as the RDF triple store for the semantic registries and deployed within Tomcat 7.

Table 4.1 shows the characteristics of each WSC'08 dataset. The number of services and concepts in the ontology of each dataset are shown in columns *#Serv.* and *#Con.* respectively. The quality of the solutions is based on the number of services and the length (i.e., number of layers) of the composition. The optimal quality of solution for each dataset (according to the WSC'08 competition) are shown in columns *#Serv.Sol.* and *Length*.

Experimentation was done using the configurations explained in Sec. 4.6 with one instance of iServe in order to measure the effect of the Discovery/Matchmaking over the whole

Table 4.1: Characteristics of the WSC'08 datasets.

Dataset	#Serv.	#Con.	#Serv.Sol.	Length
WSC'08 01	158	1,540	10	3
WSC'08 02	558	1,565	5	3
WSC'08 03	604	3,089	40	23
WSC'08 04	1,041	3,135	10	5
WSC'08 05	1,090	3,067	20	8
WSC'08 06	2,198	12,468	40	9
WSC'08 07	4,113	3,075	20	12
WSC'08 08	8,119	12,337	30	20

composition process. Results with each configuration are shown in Table 4.2. The second column shows the size (number of services) of the resulting composition graph for each dataset. The next columns show the time taken to generate the composition graph (*G. time*) in seconds and the number of *SPARQL* queries generated during that process. The last three columns show the size of the graph after the graph-based optimisations, the time of the composition search (graph optimisations + optimal A* backward search) and the number of services and length of the optimal composition found. Note that the backward optimal search does not depend on the configuration selected since it only uses the information in the composition graph.

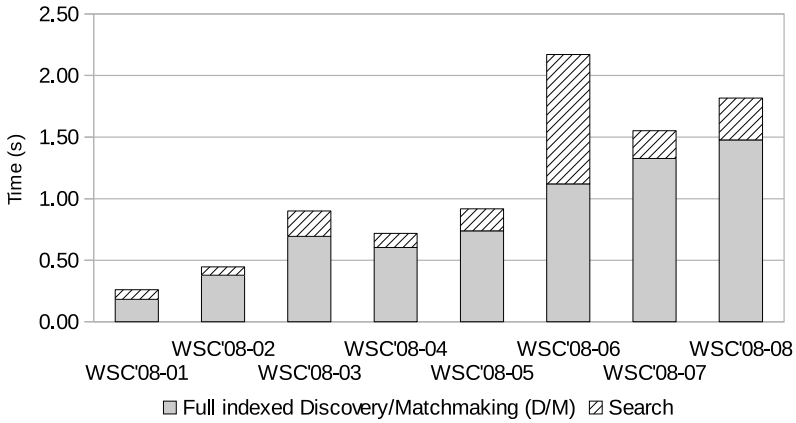

Figure 4.4: Graph generation time vs Search time for the Full Indexed Discovery/Matchmaking configuration.

The analysis of these results reveals that the discovery and matchmaking phases take most

Table 4.2: Evaluation results with different Discovery/Matchmaking (D/M) configurations with the WSC'08 datasets

	Discovery/Matchmaking (D/M)							Composition		
	1) SPARQL D/M			2) Index. D/SPARQL+Cache M		3) Full Indexed D/M				
Dataset	G. size	G. time (s)	#SPARQL	G. time (s)	#SPARQL	G. time (s)	#SPARQL	G. size (opt)	Comp. time (s)	Sol. (serv./length)
WSC'08-01	35	28.52	3256	5.67	624	0,18	0	13 (-37%)	0.08	10/5
WSC'08-02	35	63.30	7349	11.76	1830	0,38	0	13 (-37%)	0.07	5/3
WSC'08-03	105	262.80	36619	20.05	3184	0.69	0	40 (-38%)	0.21	40/23
WSC'08-04	44	136.20	13828	21.12	3481	0.60	0	25 (-57%)	0.12	10/5
WSC'08-05	97	333.60	41148	26.05	4417	0.74	0	52 (-54%)	0.18	20/8
WSC'08-06	189	1051.20	93682	48.21	8511	1.12	0	75 (-40%)	1.05	42/7
WSC'08-07	124	1183.20	120881	35.76	6376	1.33	0	70 (-56%)	0.23	20/12
WSC'08-08	121	1656.00	89518	78.00	15844	1.48	0	58 (-48%)	0.34	30/20

of the time of the composition, even using the optimal configuration (*Full Indexed D/M*) to avoid the latency of the *SPARQL* queries. This is graphically represented in Fig. 4.4. This figure shows the overall composition time for each dataset including the relative time of the *Full Indexed D/M* (blue bar) and the *Composition Search* (red bar). The *Full Indexed D/M* takes 77% of the total composition time on average. This percentage is even higher (about 99%) if the discovery and matchmaking are not optimised using indexes and cache. In other words, as anticipated by the complexity analysis presented earlier, discovery and matchmaking are responsible for the majority of the computation that needs to be performed to compose services. Optimising both phases is thus fundamental.

The comparison of the scalability of the three configurations with respect to the number of services is shown in Fig. 4.5. As can be seen, directly querying the backend (see *SPARQL* D/M), which is the approach followed by most discovery engines, rapidly becomes prohibitively slow taking 1,656 seconds (i.e., 27.6 min) in the largest dataset. Indeed, the generation of the composition graph requires computing every semantic match between all inputs and outputs as well as discovering relevant services at each layer. Doing so leads to issuing thousands of *SPARQL* queries. This can be dramatically improved using a discovery index and a local cache for the matchmaking system as can be seen in the second configuration. In this case, almost every composition is calculated in less than a minute. The generated *SPARQL* queries in this case are reduced by up to 91% (for the WSC'08-3 dataset) leading to a significant performance improvement. Although such an improvement can be enough to solve the smaller datasets in a few seconds, the latency of the *SPARQL* queries still remains a bottleneck for bigger datasets like the WSC'08-08 dataset that still require evaluating 15,844 *SPARQL* queries for generating the composition graph in 78 seconds. Our tests show, however, that the full indexed configuration allows solving the largest problems very fast by

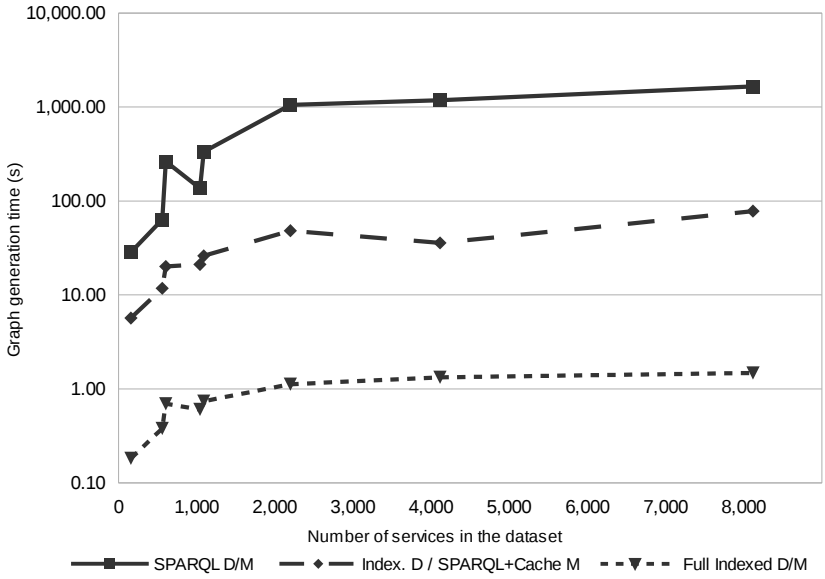

Figure 4.5: Composition time for different configurations.

avoiding the evaluation of *SPARQL* queries at composition time. This configuration entails the derived need for service registries to additionally calculate and maintain the indexes. Doing so, nonetheless, enables performing very efficient composition over remote 3rd party controlled service registries akin to what can be obtained by the fastest composition engines in the unrealistic scenarios where all services are available and pre-loaded in memory. Additionally, indeed, using those indexes allows service registries to provide highly efficient discovery for a controlled set of queries, while retaining the ability to offer fully flexible yet less efficient discovery support.

We have also evaluated our framework with the WSC'09-10 datasets. Results show a similar scalability behaviour with the number of services for each configuration. Moreover, our approach is able to solve all the datasets with optimal results, which are shown at https://wiki.citius.usc.es/composit:wsc09.

4.8 Conclusions

In this paper we have presented a theoretical analysis of service composition in terms of its dependency with service discovery. Driven by this analysis we have defined a formal integrated graph-based composition framework anchored on the integration of service discovery and matchmaking within the composition process. We have devised a reference implementation of this framework on the basis of two pre-existing separate components, namely iServe and ComposIT. This reference implementation has been used to empirically study the impact of discovery and matchmaking on service composition, and we have provided three different configurations with varying performance. Our empirical analysis shows that, indeed, typical approaches followed by discovery engines cannot serve as a suitable basis to support efficient service composition as they lead to prohibitive execution times. We have also shown, though, that with the adequate interface granularity and indexing, discovery engines can support highly efficient composition akin to that obtained by the fastest composition engines without having to assume to local availability and in-memory preloading of service registries.

This work proves the scalability and flexibility of our proposal and provides insights on how integrated composition systems can be designed in order to achieve good performance in real scenarios, where service registries and composition frameworks are likely to be distributed and controlled by diverse organisations.

Chapter 5

Hybrid Optimization Algorithm for Large-Scale QoS-Aware Service Composition

In chapter 4 we presented an integrated framework for discovery and composition of semantic Web services. This framework provides the foundations for performing efficient automatic I/O discovery and composition minimizing the total number of services. Here we present an extension of this framework to support the optimization of both the size of the compositions and the end-to-end QoS. To this end, we present an extension to the *Service Model* to include support for QoS attributes and we define a formal QoS algebra for computing the end-to-end QoS in composition graphs for any total-ordered QoS attribute. The inclusion of QoS features requires also extensions in the computation of the *Service Match Graph* and new algorithms to optimize both the number of services and the end-to-end QoS of the solutions. Concretely, the previous version of the algorithm used to compute the *Service Match Graph* stops when all the outputs of the request are matched by the ouputs of the selected services. Although this condition is enough to find the smallest composition (in terms of number of services and length of the composition), it is not enough to guarantee the optimal QoS since this graph does not contain the whole set of relevant services. Thus, we extend this algorithm to compute the whole set of service candidates. We also extend the optimizations presented in the previous chapter to take into account QoS and we introduce a new step in the optimization pipeline to prune suboptimal QoS services (i.e., services that cannot be part of the optimal solution). In

order to extract the optimal composition from this new extended graph, minimizing both the end-to-end QoS and the number of services, we develop a novel hybrid local-global search algorithm that combines i) a fast local search to obtain a near-optimal number of services while satisfying the optimal end-to-end QoS with ii) a global seach that can improve the solution obtained with the local search strategy by performing an exhaustive combinatorial search to select the composition with the minimum number of services for the optimal QoS. A comprehensive validation with the datasets of the Web Service Challenge 2009 and with random generated datasets is also provided.

All these contributions are encompassed in the following publication:

Pablo Rodríguez-Mier[1], Manuel Mucientes[1], and Manuel Lama[1]. Hybrid Optimization Algorithm for Large-Scale QoS-Aware Service Composition. *IEEE Transactions on Services Computing*, 2015. IEEE. ISSN: 1939-1374. DOI: 10.1109/TSC.2015.2480396. URL: http://dx.doi.org/10.1109/TSC.2015.2480396.

5.1 Abstract

In this paper we present a hybrid approach for automatic composition of Web services that generates semantic input-output based compositions with optimal end-to-end QoS, minimizing the number of services of the resulting composition. The proposed approach has four main steps: 1) generation of the composition graph for a request; 2) computation of the optimal composition that minimizes a single objective QoS function; 3) multi-step optimizations to reduce the search space by identifying equivalent and dominated services; and 4) hybrid local-global search to extract the optimal QoS with the minimum number of services. An extensive validation with the datasets of the Web Service Challenge 2009-2010 and randomly generated datasets shows that: 1) the combination of local and global optimization is a general and powerful technique to extract optimal compositions in diverse scenarios; and 2) the hybrid strategy performs better than the state-of-the-art, obtaining solutions with less services and optimal QoS.

[1]Centro Singular de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela.

5.2 Introduction

Web services are self-describing software applications that can be published, discovered and invoked accross the Web using standard technologies [7]. The functionality of a Web service is mainly determined by the functional properties that describe their behaviour in terms of its inputs, outputs, and also possibly additional descriptions that the services may have, such as preconditions and effects. These four characteristics, commonly abbreviated IOPEs, allow the composition and aggregation of Web services into composite Web services that achieve more complex functionalities and, therefore, solve complex user needs that cannot be satisfied with atomic Web services. However, compositions should go beyond achieving a concrete functionality and take into account other requirements such as Quality-of-Service (QoS) to generate also compositions that fit the needs of different contexts. The QoS determines the value of different quality properties of services such as response time (total time a service takes to respond to a request) or throughput (number of invocations supported in a given time interval), among others characteristics. These properties apply both to single services and to composite services, where each individual service in the composition contributes to the global QoS. For composite services this implies that having many different services with similar or identical functionality, but different QoS, may lead to a large amount of possible compositions that satisfy the same functionality with different QoS but also with a different number of services.

However, the problem of generating automatic compositions that satisfy a given request with an optimal QoS is a very complex task, specially in large-scale environments, where many service providers offer services with similar functionality but with different QoS. This has motivated researchers to explore efficient strategies to generate QoS-aware Web service compositions from different perspectives [92, 114]. But despite the large number of strategies proposed so far, the problem of finding automatic compositions that minimize the number of services while guaranteeing the optimal end-to-end QoS is rarely considered. Instead, most of the work has focused on optimizing the global QoS of a composition or improving the execution time of the composition engines. An analysis of the literature shows that only a few works take into consideration the number of services of the resulting optimal QoS compositions. Some notable examples are [4, 28, 46, 130]. Although most of these composition engines are quite efficient in terms of computation time, none of them are able to effectively minimize the total number of services of the solution while keeping the optimal QoS.

The ability to provide not only optimal QoS but also an optimal number of services is

specially important in large-scale scenarios, where the large number of services and the possible interactions among them may lead to a vast amount of possible solutions with different number of services but also with the same optimal QoS for a given problem. Moreover, there can be situations where certain QoS values are missing or cannot be measured. Although the prediction of QoS can partially alleviate this problem [135], it is not always possible to have historical data in order to build statistical models to accurately predict missing QoS. In this context, optimizing not only the available QoS but also the number of services of the composition may indirectly improve other missing properties. This has important benefits for brokers, customers and service providers. From the broker point of view, the generation of smaller compositions is interesting to achieve manageable compositions that are easier to execute, monitor, debug, deploy and scale. On the other hand, customers can also benefit from smaller compositions, specially when there are multiple solutions with the same optimal end-to-end QoS but different number of services. This is even more important when service providers do not offer fine-grained QoS metrics, since decreasing the number of services involved in the composition may indirectly improve other quality parameters such as communication overhead, risk of failure, connection latency, etc. This is also interesting from the perspective of service providers. For example, if the customer wants the cheapest composition, the solution with fewer services from the same provider may also require less resources for the same task.

However, one of the main difficulties when looking for optimal solutions is that it usually requires to explore the complete search space among all possible combinations of services, which is a hard combinatorial problem. In fact, finding the optimal composition with the minimum number of services is NP-Hard (see Appendix A). Thus, achieving a reasonable trade-off between solution quality and execution time in large-scale environments is far from trivial, and hardly achievable without adequate optimizations.

In this paper we focus on the automatic generation of semantic input-output compositions, minimizing both a single QoS criterion and the total number of services subject to the optimal QoS. The main contributions are:

- A multi-step optimization pipeline based on the analysis of non-relevant, equivalent and dominated services in terms of interface functionality and QoS.

- A fast local search strategy that guarantees to obtain a near-optimal number of services while satisfying the optimal end-to-end QoS for an input-output based composition request.

- An optimal combinatorial search that can improve the solution obtained with the local search strategy by performing an exhaustive combinatorial search to select the composition with the minimum number of services for the optimal QoS.

We tested our proposal using the Web Service Challenge 2009-2010 datasets and, also, a different randomly generated dataset with a variable number of services. The rest of the paper is organized as follows: Sec. 5.5 introduces the composition problem, Sec. 5.6 describes the proposed approach, Sec. 5.7 presents the results obtained, and Sec. 5.8 gives some final remarks.

5.3 Related Work

Automatic composition of services is a fundamental and complex problem in the field of Service Oriented Computing, which has been approached from many different perspectives depending on what kinds of assumptions are made [36, 45, 92, 114]. AI Planning techniques have been traditionally used in service composition to generate valid composition plans by mapping services to actions in the planning domain [5, 24, 51, 87, 108, 110]. These techniques work under the assumption that services are complex operators that are well defined in terms of IOPEs, so the problem can be translated to a planning problem and solved using classical planning algorithms. Most of these approaches have been mainly focused on exploiting semantic techniques [5,43,110] and developing heuristics [5,51,70] to improve the performance of the planners. As a result, and partly given by the complexity of generating satisfiable plans in the planning domain, these approaches do not generate neither optimal plans (minimizing the number of actions) nor optimal QoS-aware compositions.

Other approaches have studied the QoS-aware composition problem from the perspective of Operation Research, providing interesting strategies for optimal selection of services and optimizing the global QoS of the composition subject to multiple QoS constraints. A common strategy is to reduce the composition problem to a combinatorial Knapsack-based problem, which is generally solved using constraint satisfaction algorithms (such as Integer Programming) [8, 16, 131, 133, 136] or Evolutionary Algorithms [21, 119]. Some relevant approaches are [8, 133]. In [133] the authors present AgFlow, a QoS middleware for service composition. They analyze two different methods for QoS optimization, a local selection and a global selection strategy. The second strategy is able to optimize the global end-to-end QoS of the composition using a Integer Linear Programming method, which performs better

than the suboptimal local selection strategy. Similarly, in [8] the authors propose a hybrid QoS selection approach that combines a global optimization strategy with local selection for large-scale QoS composition. The assumption made by all these approaches is that there is only one composition workflow with a fixed set of abstract tasks, where each abstract task can be implemented by a concrete service. Both the composition workflow and the service candidates for each abstract task are assumed to be prefined beforehand, so these techniques are not able to produce compositions with variable size.

A different category of techniques are graph-based approaches that 1) generate the entire composition by selecting and combining relevant services and 2) optimize the global QoS of the composition. These techniques usually combine variants or new ideas inspired by different fields, such as AI Planning, Operations Research or Heuristic Search, in order to resolve more efficiently the automatic QoS composition, usually for a single QoS criterion. Some relevant approaches in this category are the top-3 winners of the Web Service Challenge (WSC) 2009-2010 [4, 46, 130]. Concretely, the winners of the WSC challenge [46], presented an approach that automatically discovers and composes services, optimizing the global QoS. This approach also includes an optimization phase to reduce the number of services of the solution. Although the proposed algorithm has in general good performance, as demonstrated in the WSC, it cannot guarantee to obtain optimal solutions in terms of number of services. The other participants of the WSC have also the same limitation.

A recent and interesting approach in this category has been recently presented by Jiang et al. [47]. In this paper, the authors analyze the problem of generating top K query compositions by relaxing the optimality of the QoS in order to introduce service variability. However, the compositions are generated at the expense of worsening the optimal QoS, instead of looking first for all possible composition alternatives with the minimum number of services that guarantee the optimal QoS.

Another interesting graph-based approach has been presented in [28]. In this paper, the authors propose a service removal strategy that detects services that are redundant in terms of functionality and QoS. Results show that service removal techniques can be very effective to reduce the number of services before extracting the final composition, as anticipated by other similar approaches [14, 96, 120]. However, some important limitations of this work are: 1) The QoS is not always optimal, since the graph generated for the composition is not complete as it does not contain all the relations between services (it is acyclic) and 2) although the redundancy removal is an effective technique that can be used also to prune the search space,

this strategy itself cannot provide optimal results in terms of number of services, and it should be combined with exhaustive search to improve the solutions obtained.

In summary, despite the large number of approaches for automatic QoS-aware service composition there is a lack of efficient techniques that are not only able to optimize the global end-to-end QoS, but also effectively minimize the number of services of the composition. This paper aims to provide an efficient graph-based approach that uses a hybrid local-global optimization algorithm in order to find optimal compositions both in terms of single QoS criteria and in terms of minimum number of services.

5.4 Motivation

The aim of the automatic service composition problem, as considered in this paper, is to automatically select the best combination of available QoS-aware services in a way that can fulfil a user request that otherwise could not be solved by just invoking a single, existing service. This request is specified in terms of the information that the user provides (inputs), and the information it expects to obtain (outputs). The resulting composition should meet this request with an optimal, single criterion end-to-end QoS and using as less services as possible.

A motivating example of the problem is shown in Fig. 5.1. The figure represents a graph with all the relevant services for a request R where the inputs are $\{$*ont3:IPAddress*, *ont2:MerchantCode*$\}$ and the output is $\{$*xsd:boolean*$\}$. The goal of this example is to obtain a composition to predict whether a business transaction is fraudulent or not. Each service (associated to a response time QoS) is represented by squares. Inputs and outputs are represented by circles. The graph also contains edges connecting outputs and inputs. These edges represent valid semantic *matches* whenever an output of a service can be passed as an input of a different service. As can be seen, there are some inputs (*ont1:Location*,*ont3:Payment*) that can be matched by more than one output, so there are many different ways to combine services to achieve the same goal.

Although finding the proper combination of services in terms of their inputs/outputs is essential to generate a solution, it is not enough to obtain good compositions, since there can exist different combinations of services with different QoS. Moreover, many different combinations of services may produce compositions with a different number of services but the same end-to-end QoS. For example, in Fig. 5.1 we can select *WS E-Payment* service or the *Secure Payment* service to process the electronic payment. However, the second service has a

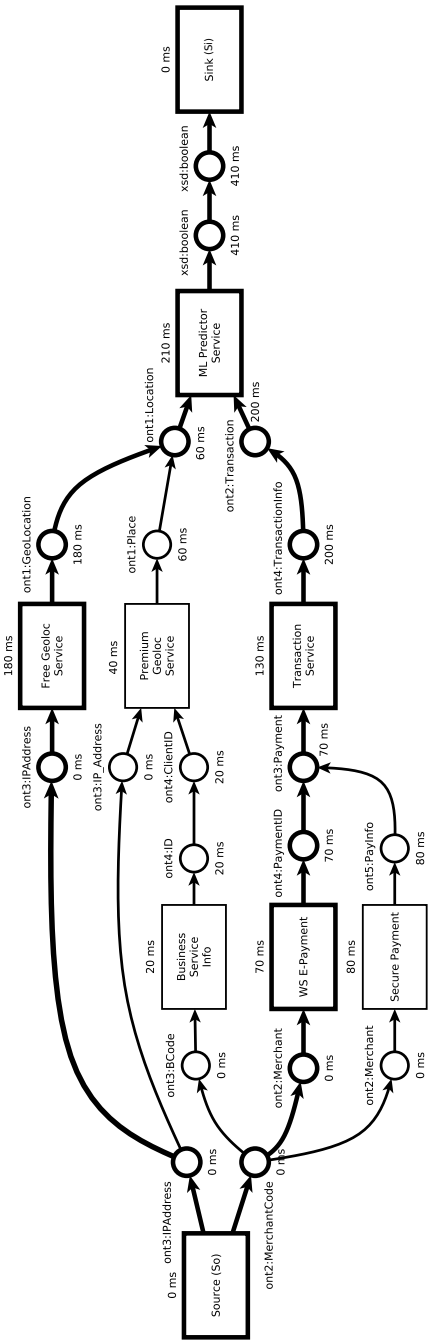

Figure 5.1: Example of a *Service Match Graph* for a request with inputs *ont3:IPAddress* and *ont2:MerchantCode* and an output *xsd:boolean* to predict whether a business transaction is fraudulent or not. The optimal solution (*Service Composition Graph*), with an overall response time of 410 ms and 4 services (excluding *So* and *Si*) is highlighted.

higher response time. Using this leads to a sub-optimal end-to-end QoS of 420 ms. However, there are other situations where the selection of different services leads to compositions with different size but same end-to-end QoS. For example, both *Free Geoloc Service* or the *Premium Geoloc Service* can be selected to translate an *IP* to a *Location*. Although the second one has a better average response time (40 ms), it requires an additional service to obtain the *ClientID* for verification purposes. However, selecting the *Premium Geoloc Service* or the *Free Geoloc Service* does not have an impact on the global QoS, since the *ML Predictor Service* has to wait longer to obtain the *Transaction* parameter (200 ms), but it has an impact on the total number of services of the solution.

The goal of this paper is to automatically generate, given a composition request, a graph like the one represented in Fig. 5.1 as well as to extract the optimal end-to-end QoS composition with the minimum number of services from that graph.

5.5 Problem Formulation

We herein formalize the main concepts and assumptions regarding the composition model used in our approach, which consists of a semantic, graph-centric representation of the service composition. These concepts are captured in three main models: 1) a service model, which is used to represent services and define how services can be connected or matched to generate composite services; 2) a graph-based composition model, which is used to represent both service interactions and compositions; and 3) a QoS computation model, which provides the operators required to compute the global QoS in a graph-based composition.

5.5.1 Semantic Service Model

The automatic composition of services requires a mechanism to select appropiated services based on their functional descriptions, as well as to automatic match the services together by linking their inputs and outputs to generate executable data-flow compositions. To this end, we introduce here the main concepts that we use in this paper to support the automatic generation of compositions. This model is an extension of a previous model used in [100] to include QoS properties.

Definition 19. *A Composition Request R is defined as a tuple* $R = \{I_R, O_R\}$*, where* I_R *is the set of provided inputs, and* O_R *the set of expected outputs. Each input and output is related to a semantic concept from the set C of the concepts defined in an ontology Ont* ($In_w, Out_w \subseteq C$).

We say that a composition satisfies the request R if it can be invoked with the inputs in I_R and returns the outputs in O_R.

Definition 20. *A Semantic Web Service (hereafter "service") can be defined as a tuple $w = \{In_w, Out_w, Q_w\} \in W$ where In_w is a set of inputs required to invoke w, Out_w is the set of outputs returned by w after its execution, $Q_w = \{q_w^1, \ldots, q_w^n\}$ is the set of QoS values associated to the service, and W is the set of all services available in the service registry.*

Each input and output is related to a semantic concept from the set C of the concepts defined in an ontology Ont ($In_w, Out_w \subseteq C$). Each QoS value $q_w^i \in Q_w$ has a concrete type associated to a set of valid values Q. For example, the QoS values of a service w with two different measures, an average response time of 20 ms and an average throughput of 1000 invocations/second, is represented as $Q_w = \{20ms, 1000\ inv/s\}$, where $20ms \in Q_{RT}$ and $1000\ inv/s \in Q_{TH}$.

Semantic inputs and outputs are used to compose the functionality of multiple services by matching their inputs and outputs together. In order to measure the quality of the match, we need a matchmaking mechanism that exploits the semantic I/O information of the services. The different matchmaking degrees that are contemplated are *exact*, *plugin*, *subsumes* and *fail* [76].

Definition 21. *Given $a, b \in C$, degree(a,b) returns the degree of match between both concepts (exact, plugin, subsume or fail), which is determined by the logical relationship of both concepts within the Ontology.*

Definition 22. *Given $a, b \in C$, match(a,b) holds if $degree(a,b) \neq fail$.*

In order to determine which concepts are matched by other concepts, we define a matchmaking operator "$\otimes$" that given two sets of concepts $C_1, C_2 \subseteq C$, it returns the concepts from C_2 matched by C_1.

Definition 23. *Given $C_1, C_2 \subseteq C$, we define "$\otimes : C \times C \rightarrow C$" such that $C_1 \otimes C_2 = \{c_2 \in C_2 | match(c_1, c_2), c_1 \in C_1\}$.*

We can use the previous operator to define the concepts of full and partial matching between concepts.

Definition 24. *Given $C_1, C_2 \subseteq C$, a full matching between C_1 and C_2 exists if $C_1 \otimes C_2 = C_2$, whereas a partial matching exists if $C_1 \otimes C_2 \subset C_2$.*

Definition 25. *Given a set of concepts $C' \subseteq C$, a service $w = \{In_w, Out_w\}$ is invokable if $C' \otimes In_w = In_w$, i.e., there is a full match between the provided set of concepts C' and In_w, so the information required by w is fully satisfied.*

This internal model used by the algorithm, which captures the core components required to perform semantic matchmaking and composition of services, is agnostic to how semantic services are represented. Thus, the algorithm is not bound to any concrete service description. Concretely, different service descriptions can be handled by the algorithm through the use of iServe importers for OWL-S, WSMO-lite, SAWSDL or MicroWSMO. For further details see [81].

5.5.2 Graph-Based Composition Model

In a nutshell, a data-flow composition of services can be seen as a set of services connected together through their inputs and output, using the semantic model defined before, in a way that every service in the composition is invocable and the invocation of each service in the composition can *transform* a set of inputs into a set of outputs. These concepts can be naturally captured by graphs, where the vertices represent inputs, outputs and services, and the edges represent semantic matches between inputs and outputs. Here we define the notion of *Service Match Graph* and *Service Composition Graph*. The *Service Match Graph* is a graph that captures all the existent dependencies (matches) between all the relevant services for a composition request. The *Service Composition Graph* is a particular case of the *Service Match Graph* that represents a composition contained in the *Service Match Graph*.

The *Service Match Graph* represents the space of all possible valid solutions for a composition request R, and it is defined as a directed graph $G_S = (V, E)$, where:

- $V = W_R \cup I \cup O \cup \{So, Si\}$ is the set of vertices of the graph, where $W_R \subseteq W$ is the set of relevant services, I is the set of inputs and O is the set of outputs. Si and So are two special services, called *Source* and *Sink* defined as $So = \{\emptyset, I_R\}$, $Si = \{O_R, \emptyset\}$.
- $E = IW \cup WO \cup OI$ is the set of edges in the graph where:
  - $IW \subseteq \{(i_w, w) \mid i_w \in I \wedge w \in W\}$ is the set of input edges, i.e., edges connecting input concepts to their services.
  - $WO \subseteq \{(w, o_w) \mid w \in W \wedge o_w \in O\}$ is the set of output edges, i.e., edges connecting services with their output concepts.

– $OI \subseteq \{(o_w, i_{w'}) \mid o_w, i_{w'} \in (I \cup O) \wedge match(o_w, i_{w'})\}$ is the set of edges that represent a semantic match between an output of w and an input of w'.

There are also some restrictions in the edge set to ensure that each input/output belongs to a single service:

- $\forall i \in I \; d^+_{G_S}(i) = 1 \wedge ch_{G_S}(i) = \{w\}, w \in W$ (each input has only one outgoing edge which connects the input with its service)
- $\forall o \in O, d^-_{G_S}(o) = 1 \wedge par_{G_S}(o) = \{w\}, w \in W$ (each output has only one incoming edge which connects the output with its service)

Function $d^+_{G_S}(v)$ returns the outdegree of a vertex $v \in G_S$ (number of children vertices connected to v), whereas $d^-_{G_S}(v)$ returns the indegree of a vertex v (number of parent vertices connected to v). The functions $ch_G(v)$ and $par_G(v)$ are the functions that returns the children vertices of v and the parent vertices of $v \in G_S$, respectively.

Fig. 5.1 shows an example of a *Service Match Graph* where each service is associated with its average response time. As can be seen, this graph contains many different compositions since there are inputs in the graph that can be matched by the outputs of different services. For example, the parent nodes of the input *ont1:Location* of the service *ML Service Predictor* (par_G(*ont1:Location*)) in Fig. 5.1 are *ont1:GeoLocation* and *ont1:Place*, so the input is matched by two outputs $d^-_{G_S}$(*ont1:Location*) $= 2$.

A *Service Composition Graph*, denoted as $G_C = (V, E)$, represents a solution for the composition request where each input is exactly matched by one output. Formally, it is a subgraph of *Service Match Graph* ($G_C \subseteq G_S$) that satisfies the following conditions:

- $\forall i \in I, d^-_{G_C}(i) = 1$ (each input is strictly matched by one output)
- G_C is a Directed Acyclic Graph (DAG)

These conditions are important in order to guarantee that a solution is valid, i.e, each input is matched by an output of a service and each service is invocable (all inputs on the composition are matched with no cyclic dependencies). This definition of service composition is language-agnostic, so the resulting DAG is a representation of a solution for the composition problem which can be translated to a concrete language, such as OWL-S or BPEL.

5.5.3 QoS Computation Model

Before looking for optimal QoS service compositions, we need first to define a model to work with QoS over compositions of services which allow us to determine the best QoS that can be achieved for a given composition request on a service repository. When many services are chained together in a composition, the QoS of each individual service contributes to the global QoS of the composition. For example, suppose we want to measure the total response time of a simple composition with two services chained in sequence. The total response time is calculated as the sum of the response time of each service in the composition. However, if the composition has two services in parallel, the total time of the composition is given by the slowest services. Thus, the calculation of the QoS of a composition depends on the type of the QoS and on the structure of the composition.

In order to define the common rules to operate with QoS values in composite services, many approaches use a QoS computation model based on workflow patterns [22], which is adequate to measure the QoS of control-flow based compositions. However, this paper focuses on the automatic generation of optimal QoS-aware compositions driven by the data-flow analysis of the service dependencies (input-output matches) that are represented as a *Service Match Graph*.

In this section we explain the general graph-centric QoS computation model that we use, based on the *path algebra* defined in [25]. This model is better suited to compute QoS values in a *Service Match Graph*, which, for extension, is also applicable to the particular case of the *Service Composition Graph*.

Definition 26. *$(Q, \oplus, \ominus, \preceq)$ is a QoS algebraic structure to operate with a set of QoS values, denoted as Q. This set is equipped with the following elements:*

- $\oplus : Q \times Q \to Q$ *is a closed binary operation for aggregating QoS values*
- $\ominus : Q \times Q \to Q$ *is a binary operation for subtracting QoS values*
- $\preceq$ *is a total order relation on Q*

This algebraic structure has the following properties:

1. Q is closed under $\oplus$ (any aggregation of two QoS values always returns a QoS value)
2. The set Q contains an identity element e such that $\forall a \in Q, a \oplus e = e \oplus a = a$

3. The set Q contains a zero element ϕ such that $\forall a \in Q, \phi \oplus a = a \oplus \phi = \phi$

4. The operator $\oplus$ is associative

5. The operator $\oplus$ is monotone for $\preceq$ (preserves order). This implies that $\forall a,b,c \in Q, a \preceq b \Leftrightarrow a \oplus c \preceq b \oplus c$

6. The operator $\ominus$ is the inverse of $\oplus$: $a \ominus b = c \Leftrightarrow a = c \oplus b$

Table 5.1 shows an example of the concrete elements in this algebra. Note that, for the sake of brevity, only the response time and throughput operators are represented in Table 5.1. However, other QoS properties such as cost, availability, reputation, etc, can also be defined by instantiating the corresponding operators. We denote Q_{RT} the set of QoS values for response time (in milliseconds), Q_{TH} the set of QoS values for throughput (invocations/second). The total order comparator $\preceq$ is required to be able to order and compare different QoS values. Given two QoS values $a,b \in Q$, $a \preceq b$ means that a is equal or ***better*** than b, whereas $b \preceq a$ means that a is equal or ***worse*** than b. The order depends on the concrete comparator defined on Q. For example, Q_{RT} uses the comparator $\leq$ to order the response time, so $a,b \in Q_{RT}$, $a \preceq b \Leftrightarrow a \leq b$. For example, given two response times $10ms, 20ms \in Q_{RT}$, $10ms \prec 20ms$ ($10ms$ is better than $20ms$) since $10ms < 20ms$. However, Q_{TH} uses the comparator $\geq$, so $a,b \in Q_{TH}$, $a \preceq b \Leftrightarrow a \geq b$. For example, given two throughput values 10 *inv/s*, 20 *inv/s* $\in Q_{TH}$, 20 *inv/s* $\prec$ 10 *inv/s* (20 *inv/s* is better than 10 *inv/s*) since 20 *inv/s* $>$ 10 *inv/s*. This order relation also affects the behavior of the min and max functions. The min function always selects the *best* QoS value, whereas the max function always selects the *worst* QoS value.

Table 5.1: QoS algebra elements for response time and throughput

QoS (Q)	$a \oplus b$	$a \ominus b$	e	ϕ	**Order ($\preceq$)**
$Q_{RT} = \mathbb{R}_{\geq 0} \cup \{\infty\}$	a + b	a - b	0	∞	$\leq$
$Q_{TH} = \mathbb{R}_{\geq 0} \cup \{\infty\}$	min(a, b)	min(a,b)	∞	0	$\geq$

Definition 27. *$F_Q(w) : W \rightarrow Q$ is a function that given a service $w \in W$, it returns its corresponding QoS value from Q_w with type Q. This function can be seen as a function to measure the QoS of a service.*

For example, in Fig. 5.1, $F_{Q_{RT}}(\textit{Trans. Service}) = 130ms$.

Definition 28. *$V_Q(w) : W \rightarrow Q$ is a function that given a service w, it returns its aggregated QoS value. This is defined as:*

$$V_Q(w) = \begin{cases} \max\limits_{\forall i \in In_w} (V_Q^{in}(i)) \oplus F_Q(w) & \text{if } In_w \neq \emptyset \\ F_Q(w) & \text{if } In_w = \emptyset \end{cases} \tag{5.1}$$

Informally, this function calculates the aggregated QoS of a service by taking the worst value of the QoS of its inputs plus the current QoS value of the service itself. Taking for example the service *Premium Geoloc Service* from Fig. 5.1, $V_{Q_{RT}}(\textit{Premium Geoloc Service})$ is computed as $max(V_{Q_{RT}}^{in}(\textit{ont3:IP\_Address}), V_{Q_{RT}}^{in}(\textit{ont4:ClientID})) \oplus 40ms$, which is $max(0ms, 20ms) \oplus 40ms = 60ms$ (see Def. 30).

Definition 29. *$V_Q^{out}(o_w) : O \rightarrow Q$ is a function that given an output of a service w, $o_w \in O$, it returns its aggregated QoS value. The aggregated QoS of an output is equal to the aggregated QoS of a service. Thus, it is defined as:*

$$V_Q^{out}(o_w) = V_Q(w) \tag{5.2}$$

For example, the aggregated QoS of the output *ont1:Place* ($V_{Q_{RT}}^{out}(\textit{ont1:Place})$) is equal to the aggregated QoS of its service *Premium Geoloc Service* ($V_{Q_{RT}}(\textit{Premium Geoloc Service})$), which is equal to *60ms*.

Definition 30. *$V_Q^{in}(i_w) : I \rightarrow Q$ is a function that given an input of a service w, $i_w \in I$, it returns its optimal aggregated QoS value. This function is defined as:*

$$V_Q^{in}(i_w) = \begin{cases} \phi & \text{if } d_{G_S}^-(i_w) = 0 \\ V_Q^{out}(o_{w'}), o_{w'} \in par_G(i_w) & \text{if } d_{G_S}^-(i_w) = 1 \\ \min\limits_{\forall o_{w'} \in par_G(i_w)} (V_Q^{out}(o_{w'})) & \text{if } d_{G_S}^-(i_w) > 1 \end{cases} \tag{5.3}$$

Given an input $i_w \in In_w$ of a service w, this function returns the accumulated QoS for that input. If the evaluated input is not matched by any output ($d_{G_S}^-(i_w) = 0$), then the accumulated QoS of the input is undefined. If the evaluated input is matched by just one output ($d_{G_S}^-(i_w) = 1$), then its accumulated QoS value is equal to the accumulated QoS of that output. If the evaluated input can be matched by more than one output ($d_{G_S}^-(i_w) > 1$), i.e., there are many

services that can match that input, then its accumulated QoS value is computed by selecting the optimal (best) QoS.

For example, the optimal aggregated QoS of the input *ont3:Payment* from *Transaction Service* ($V^{in}_{Q_{RT}}$(*ont3:Payment*)) is calculated as $\min(V^{out}_{Q_{RT}}$(*ont3:PaymentID*)$, V^{out}_{Q_{RT}}$(*ont5:PayInfo*)) $= 70ms$.

Definition 31. *We define $V^G_Q(g) : G \rightarrow Q$ as a function that given a Service Match Graph $g = (V,E)$, it returns its optimal aggregated QoS value. This is defined as:*

$$V^G_Q(g) = V_Q(Si), S_i \in V \tag{5.4}$$

Basically, the optimal QoS of a *Service Match Graph* G_S corresponds with the optimal aggregated QoS of its service $Si \in G_S$.

5.5.4 Composition Problem

Given a composition request $R = \{I_R, O_R\}$, a set of semantic services W, a semantic model and a QoS algebra, the composition problem considered in this paper consists of generating the *Service Match Graph* G_S and selecting a composition graph $G_C \subset G_S$ such that:

1. $\forall G'_C, V^G_Q(G_C) \leq V^G_Q(G'_C)$, i.e., the composition graph has the best possible QoS
2. $W_R \subseteq V, |W_R|$ is minimized (the composition graph contains the minimum number of services)

5.6 Composition Algorithm

On the basis of the formal definition of the automatic QoS-aware composition problem, in this section we present our hybrid approach strategy for automatic, large-scale composition of services with optimal QoS, minimizing the services involved in the composition. The approach works as follows: given a request, a directed graph with the relevant services for the request is generated. Once the graph is built, an optimal label-correcting forward search is performed in polynomial time in order to compute the global optimal QoS. This information is used later in a multi-step pruning phase to remove sub-optimal services. Finally, a hybrid local/global search is performed within a fixed time limit to extract the optimal solution from the graph. The local search returns a near-optimal solution fast whereas the global search performs an

incremental search to extract the composition with the minimum number of services in the remaining time. In this section we explain each step of the algorithm, namely: 1) generation of the *Service Match Graph*; 2) calculation of the optimal end-to-end QoS; 3) multi-step graph optimizations and 4) hybrid algorithm.

5.6.1 Generation of the Service Match Graph

Given a composition request, which specifies the inputs provided by the user as well as the outputs it expects to obtain, and a set of available services, the first step consists of locating all the relevant services that can be part of the final composition, as well as computing all possible matches between their inputs and outputs, according to the semantic model presented in Sec. 5.5.1. The output of this step is a *Service Match Graph* that contains many possible valid compositions for the request, as the one represented in Fig. 5.2. In a nutshell, the generation of the graph is calculated by selecting all invocable services layer by layer, starting with *So* in the first layer (the source service whose outputs are the inputs of the request) and terminating with *Si* in the last layer (the sink service whose inputs are the outputs of the request) [99].

The pseudocode of the algorithm is shown in Fig. 4. The algorithm runs in polynomial time, selecting $W_{selected} \subseteq W$ services at each step. At each layer, the algorithm finds a potential set of relevant services whose inputs are matched by some outputs generated in the previous layer using the $\otimes$ operator (L.6). Then, for each potential eligible service, the algorithm checks whether the service is invokable or not (i.e., all its inputs are matched by outputs of previous layers) by checking if all the unmatched inputs of the service are matches. All the inputs that are matched are removed from the unmatched set of inputs for the current service (L.11). If the service is invokable (has no unmatched inputs), it is selected and its outputs are added to the set of the available concepts. In case the service still has some unmatched inputs, these inputs are stored in a map to check it again in the next layer. For example, the first eligible services for the request shown in Fig. 5.2 are the services in the layer $L1$, which correspond with the services whose inputs are fully matched by I_R (the set of output concepts produced in $L0$). The second eligible services are those services (placed in $L2$) whose inputs are fully matched by the outputs of the previous layers, and so on. The algorithm stops when no more services are added to the set of selected services. Finally, *COMPUTE-GRAPH* computes all possible matches between the outputs and the inputs of the selected services. The output of this process is a complete *Service Match Graph* that can contain cycles, as the one depicted in Fig. 5.2.

Algorithm 4 Algorithm for generatig a Service Match Graph from a composition request R and a set of services W.

1: **function** SERVICEMATCHGRAPH($R = \{I_R, O_R\}, W$)
2: $C := I_R$; $W' := W$; $W_R := \{So, Si\}$
3: $unmatchedIn := [\,]$; $availCon := I_R$
4: **repeat**
5: $W_{selected} = \emptyset$
6: $W_{rel} := \{w \in W' \mid availCon \otimes In_w \neq \emptyset\}$
7: $W_{rel} := W_{rel} \setminus W_R$
8: **for all** $w_i = \{In_{w_i}, Out_{w_i}\} \in W_{rel}$ **do**
9: $U_{set} := unmatchedIn[w_i]$
10: $M_{set} := C \otimes U_{set}$
11: $unmatchedIn[w_i] := U_{set} \setminus M_{set}$
12: **if** $M_{set} = \emptyset$ **then**
13: $W_{selected} = W_{selected} \cup w_i$
14: $availCon := availCon \cup Out_{w_i}$
15: $W' := W' \setminus W_{selected}$
16: $W_R := W_R \cup W_{selected}$
17: $C := C \cup availCon$
18: $availCon := \emptyset$
19: **until** $W_{selected} = \emptyset$
20: **return** COMPUTE-GRAPH(W_R)

5.6.2 Optimal end-to-end QoS

Once the *Service Match Graph* is computed for a composition request, the next step is to calculate the best end-to-end QoS achievable in the *Service Match Graph*. The optimal end-to-end QoS can be computed in polynomial time using a shortest path algorithm to calculate the best aggregated QoS values for each input and output of the graph, i.e., the best QoS values at which the outputs can be generated and the inputs are matched. In order to compute the optimal QoS, we use a generalized Dijkstra-based label-setting algorithm computed forwards from *So* to *Si* [97], based on the algebraic model of the QoS presented in Sec. 5.5. The optimality of the algorithm is guaranteed as long as the function defined to aggregate the QoS values ($\otimes$) is monotonic, in order to satisfy the principle of optimality. A proof can be found in [111].

Fig. 5 shows the pseudocode of the generalized Dijkstra-based label-setting algorithm. The algorithm starts assigning infinite QoS cost to each input in the graph in the table *qos*.

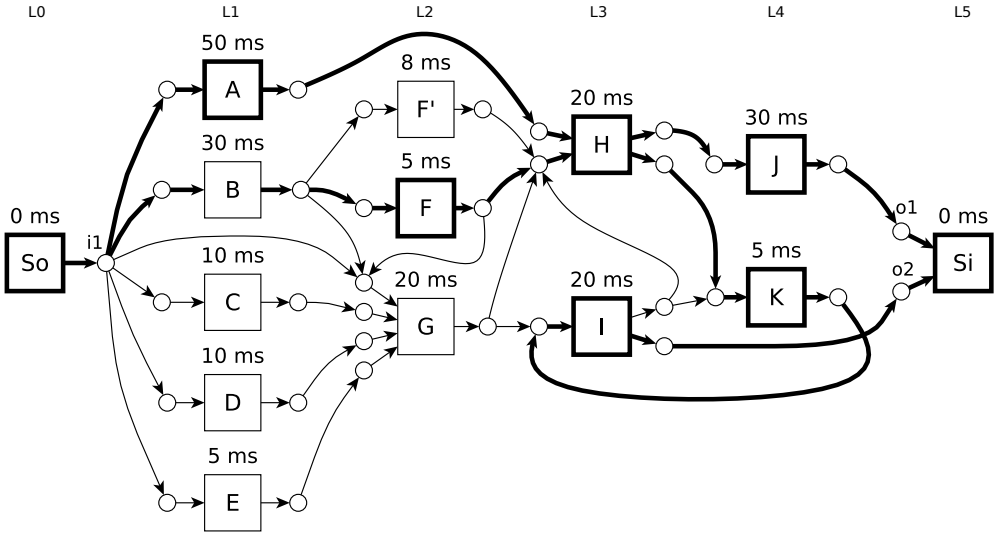

Figure 5.2: Graph example with the solution with optimal QoS and minimum number of services highlighted.

An infinite cost for an input means that the input is still not *resolved*. The first service to be processed is *So*. Each time a service w is processed from the queue, the best accumulated QoS cost of each input $i_{w'}$ matched by the outputs of the service w is recalculated. If there is an improvement (i.e., a match with a better QoS is discovered) the affected service is stored in $updated$ to recompute its new aggregated QoS. Finally, for each service $w \in updated$, we recompute its aggregated QoS using the updated values of each affected input. If the QoS has been improved, the service is added to the queue to expand it later.

5.6.3 Graph optimizations

Finding the composition with the minimum number of services is a very hard combinatorial problem which, in most cases, has a very large search space, mainly determined by the size of the *Service Match Graph*. In order to improve the scalability with the number of services, we apply a set of admissible optimizations to reduce the search space. At each pass, the algorithm analyzes different criteria to identify services that are redundant or can be substituted by better ones, so the size of the graph decreases monotonically. The different passes that are sequentially applied are: 1) elimination of services that do not contribute to the outputs of

Algorithm 5 Dijkstra-based algorithm to compute the best QoS for each input and output in the *Service Match Graph* G_S.

```
1:  function QoS-Update(G_S = {V, E})
2:      /*qos is a table indexed by inputs (i)
3:      associated to their aggregated QoS (q)*/
4:      qos[i, q] ← []
5:      for all i ∈ I, I ⊂ V do
6:          qos[i] ← φ
7:      queue ← So
8:      while queue ≠ ∅ do
9:          /* Queue sorted by aggregated QoS */
10:         w ← POP(queue)
11:         updated = {}
12:         for all o_w ∈ Out_w do
13:             for all i_w' ∈ ch_G(o_w) do
14:                 if V_Q(w) ≺ qos[i_w'] then
15:                     qos[i_w'] ← V_Q(w)
16:                     updated ← updated ∪ w'
17:         for all w ∈ updated do
18:             if cost w has been improved then
19:                 queue ←INSERT(w, queue)
20:     return qos
```

the request; 2) pruning of services that lead to suboptimal QoS; 3) combination of interface (inputs/outputs) and QoS equivalent services; and 4) replacement of interface and QoS dominated services. These optimizations are an extension of the optimizations presented in [100] to support QoS.

The **first pass** selects the set of reachable services in the *Service Match Graph*. Starting from the inputs of *Si*, it selects all those services whose outputs match any inputs of *Si*. This step is repeated with the new services until the empty set is selected. Those services that were not selected do not contribute to the expected outputs of the composition and can be safely removed from the graph.

The **second pass** prunes the services of the graph that are suboptimal in terms of QoS, i.e., they cannot be part of any optimal QoS composition. To do so, we compute the maximum admissible QoS bound for each input in the graph. In a nutshell, the maximum bound of the inputs of a service w can be calculated by selecting the maximum QoS bound among the

bounds of all inputs matched by the outputs of the service w and subtracting the QoS of w. This can be recursively defined as:

$$max^i_Q(i_w) = \begin{cases} V_Q(w) \ominus F_Q(w) & \text{if } Out_w = \emptyset \\ \max\limits_{\forall o_w, \forall i_{w'} \in ch_G(o_w)} (max^i_Q(i_{w'})) \ominus F_Q(w) & \text{if } Out_w \neq \emptyset \end{cases}$$

The value of max^i_Q for each input in the graph can be easily calculated by propagating the bounds from *Si* to *So*. For example, in Fig. 5.1, we start computing the maximum bound of the inputs of *Si* (*xsd:boolean*). Since *Si* has no outputs, $max^i_Q(\textit{xsd:boolean})$ is calculated as $V_Q(Si) \ominus F_Q(Si) = 410\ ms - 0\ ms$. Then, we select all the services whose outputs match *xsd:boolean*. In this case there is just one service, *ML Predictor Service*. The bounds of its inputs are now computed by subtracting out the $F_Q(\textit{ML Predictor Service})$ from the maximum bound of the inputs that this service matches. Since there is just one input matched (*xsd:boolean* from *Si*) whose bound is *410 ms*, we have $max^i_Q(i) = 410ms - 210ms = 200ms$ for each input i of the service. In the next step, we have three services that match the new calculated inputs (*Free Geoloc Service*, *Premium Geoloc Service* and *Transaction Service*). The maximum bounds of the inputs of these services are $200ms - 180ms = 20ms$, $200ms - 40ms = 160ms$ and $200ms - 130ms = 70ms$ respectively. Note that, since the maximum bound of *Transaction Service* is *70ms*, the service *Secure Payment* is out of the bounds (its output QoS is *80 ms*), so it can be safely pruned.

The **third and the forth pass** analyze service equivalences and dominances in the *Service Match Graph*. It is very frequent to find services from different providers that offer similar services with overlapping interfaces (inputs/outputs). In scenarios like this, it is easy to end up with large *Service Match Graph* that make very hard to find optimal compositions in reasonable time. One way to reduce the complexity without losing information is to analyze the interface equivalence and dominance between services in order to *combine* those that are equivalent, or replace those that are dominated in terms of the interface they provide and the QoS they offer. In a nutshell, we check three objectives to compare services: the *amount* of information they need to be invoked (inputs), the *amount* of information they return (outputs), and their QoS. If a set of services are equal in all objectives, they are equivalent and they can be combined into an abstract service with several possible implementations. If a service is equal in all objectives and at least better in one objective (it requires less information to be invoked, produces more information or has a better QoS), then the service *dominates* the

other service. A more detailed description of the interface and dominance optimizations is described in [100].

Note that optimizations are applied right before all semantic matches are computed in the *Service Match Graph*, since the optimizations are based on the analysis of the I/O matches among services. For this reason, they cannot be applied during the calculation of the graph (this would require to precompute in advance missing relations during the graph generation, which does not provide any benefit as this is what the *Service Match Graph* generation algorithm already does). On the other hand, optimizations are applied sequentially to save computation time, since the number of services in the graph decreases monotonically in each step. In order to take advantage of this, faster optimizations are applied first so that the slower optimizations in the pipeline can work with a reduced set of services.

5.6.4 Hybrid algorithm

Each service in the composition graph may have different services that match each input, thus there may exist multiple combinations of services that satisfy the composition request with the same or different QoS. The goal of the hybrid search is to extract good solutions from the composition graph, optimizing the total number of involved services in the composition and guaranteeing the optimal QoS. Thus, for each input we select just one service of the graph to match that input, until the best combination is found. The hybrid search performs a local search to extract a good solution and in the remaining time, it tries to improve the solution by running a global search.

Fig. 6 shows the pseudocode of the **local search** strategy. The algorithm starts with a composition graph, the inputs of the service *Si* marked as unresolved (the expected outputs of the request) and the service *Si* selected to be part of the solution. An *unresolved input* is an input that can be matched by many different outputs but no decision has been made yet. Using the list of the unresolved inputs to be matched, the method RANK-RESOLVERS returns a list of services that match any of the unresolved inputs. Services are ranked according to the number of unresolved inputs that match, so the service that matches more inputs is considered first to be part of the solution. Then, for each input that the selected service can match, the method CYCLE performs a forward search to check if resolving the selected input with that service leads to a cycle. For example, in Fig. 5.2, if we select the service K to match the input of I after having decided to resolve the input of K with the service I, we end up with an invalid composition, so K is an invalid resolver for I and it must be discarded. Once all resolvable

inputs are collected in *resolved*, the method RESOLVE creates a copy of the current graph where the inputs in *unresolved* are matched only by the selected service, i.e., any other match between any output from a different service to that input is removed from the graph. If the selected service was not already selected, then all its inputs are then marked as unresolved and a recursive call to LSBT is performed to select a new service to resolve the remaining inputs, until a solution is found. If a dead end is reached (a solution that has no services to resolve the remaining inputs without cycles) the algorithm backtracks to a previous state to try a different service (L.7).

Algorithm 6 Local search algorithm to extract a composition from a graph.

```
1:  function LOCAL-SEARCH(G_S = {V, E})
2:      return LSBT(G_S, In_Si, {Si})
3:
4:  function LSBT(G_S, unresolved, services)
5:      if unresolved = ∅ then return G_S
6:      servs ← RANK-RESOLVERS(unresolved)
7:      for each w ∈ servs do
8:          resolved ← {}
9:          matched ← Out_w ⊗ unresolved
10:         for each input ∈ matched do
11:             if ¬CYCLE(G_S, w, input) then
12:                 resolved ← resolved ∪ input
13:         if resolved ≠ ∅ then
14:             unresolved ← unresolved \ resolved
15:             if w ∉ services then
16:                 unresolved ← unresolved ∪ In_w
17:             G'_S ← RESOLVE(G_S, w, resolved)
18:             services ← services ∪ w
19:             result ← LSBT(G'_S, unresolved, services)
20:             if result ≠ fail then return result
        return fail
```

An implementation of the *CYCLE* method is provided in 7. The algorithm performs a *look-ahead* check in a breadth-first fashion to determine whether matching the selected input i with an output of the service w leads to a cyclic dependency. This is done by traversing only the resolved matches, i.e., inputs that are matched by just one output of a service, until the selected service w is reached, proving the existence of a cycle. A more memory efficient implementation of the cycle algorithm can be done using the *Tarjan's strongly connected*

components algorithm [115], stopping at the first strongly connected component detected.

Algorithm 7 Näive breadth-first-search algorithm to check whether using the service w to resolve the input $i_{w'}$ of a service w' leads to a cycle.

```
1:  function CYCLE(G_S = {V,E}, w, i_w')
2:      W_visited ← {w'}
3:      W_new ← {w'}
4:      while W_new ≠ ∅ do
5:          W_reached ← {}
6:          for all w_n ∈ W_new do
7:              for all o_{w_n} ∈ Out_{w_n} do
8:                  for all i_{w'_n} ∈ ch_{G_S}(o_{w_n}) do
9:                      if d^-_{G_S}(i_{w'_n}) = 1 ∧ w'_n ∉ W_visited then
10:                         if w'_n = w then return true
11:                         W_reached ← W_reached ∪ w'_n
12:         W_new ← W_reached
13:         W_visited ← W_visited ∪ W_new
14:     return false
```

After the local search is used to find a good solution, the **global search** is performed in the remaining time to obtain a better solution by exhaustively exploring the space of possible solutions. In a nutshell, this algorithm works as follows: Given a *Service Match Graph* G_S, with some unresolved inputs, which initially are the inputs of the service *Si*, the algorithm selects an input to be resolved and for each service candidate that can be used to resolve that input, it generates a copy of the graph G_S but with the input resolved (i.e., the selected service is the only one that matches the unresolved input). The algorithm enqueues each new graph to be expanded again, and repeats the process by extracting the graph with the minimum number of services from the queue, until it eventually finds a graph with no unresolved inputs.

Fig. 8 shows the pseudocode of the global search algorithm. The algorithm starts computing the optimal QoS of the graph with the method *QoS-UPDATE*. This method returns a key-value table $qos[i,q]$ where each key corresponds with an input i of the graph, and each value q its optimal aggregated QoS $q = V_Q^{in}(i)$. Then, the inputs of the service *Si* of the graph are added to I_{un} to mark them as unresolved (L.8). In order to minimize the number of possible candidates for each unresolved input, we compute and propagate a range of valid QoS values, called QoS bounds, and defined as an interval $[min, max]$. These bounds determine the range of valid accumulated QoS values of the outputs that can be used to match each of the unre-

solved inputs without exceeding the optimal end-to-end QoS of the final composition. The *min* value is the optimal QoS for the input, i.e., there is no output in the graph that can match the input with a lower QoS, whereas the *max* value is the maximum QoS value supported. If this bound is exceeded, the total aggregated QoS of the composition worsens. For example, in Fig. 5.1, the bounds of the input *ont4:ClientID* of the service *Premium Geoloc Service* are $[20ms, 160ms]$. If we exceed the min bound (*20 ms*), the output QoS of the service gets worse ($> 60ms$), which also affects the optimal QoS of the input *ont1:Location*. However, as long as the max bound is not exceeded ($\leq 160ms$), the optimal accumulated QoS of the *ML Predictor Service* would not be affected.

The method *COMPUTE-V_Q* is used to compute the value of the V_Q function (Eq. 5.1) using the best QoS values of inputs, stored in qos ($qos[i] = V_Q^{in}(i)$). A tuple $\langle G_S, I_{un}, qos, W_{sel} \rangle$, where G_S is the current graph, I_{un} are the unresolved inputs of G_S, qos is the best aggregated QoS values for each input in G_S and W_{sel} is the set of the selected services, defines the components of a partial solution. Each partial solution is stored in a priority queue, which is sorted by the number of services W_{sel}. This allows an exploration of the search space in a breadth-first fashion, so the solution with the minimum number of services is always expanded first. At each iteration, a partial solution is extracted from the queue to be refined (L.12). If the partial solution has no unresolved inputs, the solution is complete, and has the minimum number of services. If the partial solution still has some unresolved inputs, it is refined by selecting an unresolved input with the method *SELECT*. This method selects the input to be resolved, using a *minimum-remaining-values* heuristic. This heuristic selects always the input with less resolvers (services candidates) in order to minimize the branching factor. The list of services that can match the selected input with a total aggregated QoS value within the $[min, max]$ bound is calculated with the method *RESOLVERS*. For each valid service, the algorithm performs a *look-ahead* search to check whether using the current service to resolve the selected input leads to an unavoidable cycle. If so, the service is prematurely discarded to save computation time and space. If it does not lead to a cycle, then a copy of the graph (G'_S) with the selected input resolved is generated, and the input is also removed from the set of unresolved inputs. Using the optimal aggregated QoS values for the inputs of the graph, stored in qos, the algorithm computes the aggregated QoS value of the service w. If this value is worse than the *min* bound (COMPUTE-$V_Q(w, qos') \succ min$), then the aggregated QoS value of some inputs and outputs of the graph may be affected. Thus, a repropagation of the QoS values for each input and output is computed again over the new graph G'_S (L.22). For

example, if the *Business Service Info* increments its response time to *40 ms*, a repropagation is required to recompute the accumulated QoS of all the services that may be affected. In this case, the *Premium Geoloc Service* increments its accumulated QoS cost from *60 ms* to *80 ms*, as well as the optimal QoS of the *ont1:Location*.

Finally, if the current service is not part of the current solution, its inputs are added to the unresolved table, and a new bound for each input is computed. The *min* bound corresponds with the optimal value, which is stored in qos'. In order to compute the *max* bound, we need to *subtract* the QoS of the selected service ($F_Q(w)$) from the *max* bound of the resolved input, using the operator $\ominus$ (L.25). This new partial solution is inserted in the queue to be expanded later on.

5.7 Evaluation

In order to evaluate the performance of the proposed approach, we conducted two different experiments. In the first experiment, we evaluated the approach using the datasets of the Web Service Challenge 2009-2010 [53]. The goal of this first experiment was to evaluate the peformance and scalability of the proposed approach on large-scale service repositories. In the second experiment, we tested the algorithm with five random datasets in order to better analyze the differences of the performance between the local and the global search. All tests were executed with a time limit of 5 min. Solutions produced by our algorithm are represented as *Service Composition Graphs* (no BPEL was generated).

5.7.1 Web Service Challenge 2009-2010 datasets

The datasets of the Web Service Challenge 2009-2010 range from 572 to 15,211 services with two different QoS properties: response time and throughput. Table 5.2 shows the results obtained for each dataset and for each QoS property. The response time is the average time (measured in milliseconds) that a service takes to respond to a request. The throughput, as defined in the WSC, is the average ratio of invocations per second supported by a service.

Row *#Graph services* shows the number of services of the composition graph and *#Graph services (opt)* the number of services after applying the graph optimizations. As can be seen, the optimizations reduce, on average, by 64% the number of services in the initial composition graph. This indicates that equivalence and dominance analysis of the QoS and the functionality of services is a powerful technique to reduce the search space in large scale problems.

Algorithm 8 Global search algorithm to extract the optimal composition.

```
1:  function GLOBAL-SEARCH(G_S)
2:      qos[i,q] ← QoS-UPDATE(G_S)
3:      max ←COMPUTE-V_Q(Si,qos)
4:      W_sel ← {Si}
5:      /* I_un is a key-value table where the keys are
6:      unresolved inputs and the values their QoS bounds */
7:      for i_Si ∈ In_Si do
8:          I_un[i_Si] ← [qos[i_Si],max]
9:      /* Queue sorted by |W_sel| */
10:     queue ← INSERT(⟨G_S,I_un,qos,W_sel⟩,queue)
11:     while queue ≠ ∅ do
12:         ⟨G_S,I_un,qos,W_sel⟩ ← POP(queue)
13:         if I_un = ∅ then return G_S
14:         input ← SELECT(I_un)
15:         [min,max] ← I_un[input]
16:         for all w ∈ RESOLVERS(input, [min,max]) do
17:             if ¬CYCLE(G_S,w,input) then
18:                 G'_S ← RESOLVE(G_S,w,{input})
19:                 I'_un ← REMOVE(i,I_un)
20:                 qos' ← qos
21:                 if COMPUTE-V_Q(w,qos') ≻ min then
22:                     qos' ← QoS-UPDATE(G'_S)
23:                 if w ∉ W_sel then
24:                     W'_sel ← W_sel ∪ w
25:                     max' ← max ⊖ F_Q(w)
26:                     for i_w ∈ In_w do
27:                         min' ← qos'[i_w]
28:                         I'_un[i_w] ← [min',max']
29:             queue ← INSERT(⟨G'_S,I'_un,qos',W'_sel⟩,queue)
        return fail
```

Table 5.2: Validation with the WSC 2009-2010

		D-01	D-02	D-03	D-04	D-05
#Services in the dataset		572	4,129	8,138	8,301	15,211
Validation with Response Time						
Optimal Response Time (ms)		500	1,690	760	1,470	4,070
#Graph services		81	141	154	331	238
#Graph services (opt)		21	57	15	160	126
Local Search	#Services	5	20	10	40	32
	Time (s)	0.613	0.988	2.608	7.767	2.920
Global Search	#Services	5	20	10	-	32
	Time (s)	0.617	1.580	2.613	-	24.971
Validation with Throughput						
Optimal Throughput (inv/s)		15,000	6,000	4,000	4,000	4,000
#Graph services		81	141	154	331	238
#Graph services (opt)		10	43	90	156	69
Local Search	#Services	5	20	15	62	31
	Time (s)	0.343	1.173	1.933	8.571	2.562
Global Search	#Services	5	20	10	-	30
	Time (s)	0.345	1.246	2.085	-	119.322

Rows *Local search* and *Global search* show the number of services of the solution obtained with each respective method as well as the total amount of time spent in the search. The global search found the best solution for each dataset and for each QoS property, except for the dataset 04, where the composition with the minimum number of services could not be found due to combinatorial explosion. However, in those cases, the local search strategy is able to find an alternative solution very fast. Note also that, in many cases, the local search obtains the best solution (comparing it with the global search) except for the throughput in datasets 03 and 05.

We have compared our approach with the top-3 of the Web Service Challenge 2010 [122]. Table 5.3 shows this comparison following the same format and the same rules of the Web Service Challenge. The format, rules and other details of the challenge are described in [122]. Third and forth columns show the response time and the throughput obtained for each dataset. Note that, since all these algorithms minimize a single QoS, these values are computed by executing the algorithm twice, one for each QoS. Unfortunately, the results provided by the WSC organization in [122] show only the minimum number of services for both executions (fifth

Table 5.3: Comparison with the top 3 WSC 2010

		R.Time	Through.	Min. Serv.	Time (ms)
D-01	CAS [46]	500	15,000	**5**	78
	RUG [4]	500	15,000	10	188
	Tsinghua [130]	500	15,000	9	109
	Our approach	500	15,000	**5**	956
D-02	CAS [46]	1,690	6,000	**20**	94
	RUG [4]	1,690	6,000	40	234
	Tsinghua [130]	1,690	6,000	36	140
	Our approach	1,690	6,000	**20**	2,171
D-03	CAS [46]	760	4,000	**10**	78
	RUG [4]	760	4,000	11	234
	Tsinghua [130]	760	4,000	18	125
	Our approach	760	4,000	**10**	4,693
D-04	CAS [46]	1,470	4,000	73	156
	RUG [4]	1470	4,000	133	390
	Tsinghua [130]	1,470	4,000	133	188
	Our approach	1,470	4,000	**40**	16,338
D-05	CAS [46]	4,070	4,000	32	63
	RUG [4]	4,070	4,000	4,772	907
	Tsinghua [130]	4,070	4,000	4,772	531
	Our approach	4,070	4,000	**30**	122,242

column). Thus, the number of services obtained for both the response time and throughput is unknown, which makes it hard to compare with our results. Even so, using the same evaluation criteria, our approach obtains the optimal QoS for the response time and the throughput, and also improves the number of services in D-04 (40 vs 73) and D-05 (30 vs 32) with respect to the solutions obtained by the winner of the challenge (the minimum number of services obtained for each dataset is highlighted). The last column shows the total execution time of each algorithm. The total time includes the time spent to obtain the solution for the response time and for the throughput.

Our approach takes, in general, more time to obtain a solution. However, it should be noted that we show the best results achieved by the hybrid approach, i.e., if the global search improves the solution of the local search, we show that solution along with the time taken by the global search. Anyway, the local search always provide a first good solution very fast. For

example, as can be seen in Table 5.2, the optimal solution for D-05 has 30 services and has been obtained in 119.322 s, but the local search obtained a solution with 31 services in 2.56 s, still better than the solution with 32 services obtained by [46] (Table 5.3). Moreover, it should also be noted that the problem of finding the optimal composition with minimum number of services and optimal QoS is much harder than just optimizing the QoS objective function, which is the problem solved by the participants of the WSC 2010. Although the problem is intractable and requires exponential time, it can be optimally solved for many particular instances in a reasonable amount of time using adequate optimizations even in large datasets as shown in Tables 5.2 and 5.5. This is one of the main reasons why a combination of a local and global search can achieve good results in a wide variety of situations, in contrast with pure greedy strategies or with pure global optimization algorithms.

We also compare the results obtained with Chen et al. [28], who offer a detailed analysis of their results. This comparison is shown in Table 5.4. Solutions are compared according to their QoS and number of services. A solution is better if 1) its overall QoS is better or 2) has the same QoS but less services. The results show that our algorithm always gets same or better results. Concretely, it finds solutions with optimal QoS and less services in D-01, D-02, D-04 and D-05 (response time), and D-03 (throughput). It also finds a solution with a better QoS (4000 inv/s vs 2000 inv/s) in D-04 (throughput).

Table 5.4: Detailed comparison with [28]

		D-01	**D-02**	**D-03**	**D-04**	**D-05**
Chen et al.	R. Time	500	1,690	760	1,470	4,070
	Services	8	21	**10**	42	33
Our approach	R. Time	500	1,690	760	1,470	4,070
	Services	**5**	**20**	**10**	**40**	**32**
Chen et al.	Throughput	15,000	6,000	4,000	2,000	4,000
	Services	**5**	**20**	21	40	**30**
Our approach	Throughput	15,000	6,000	4,000	**4,000**	4,000
	Services	**5**	**20**	**10**	62	**30**

5.7.2 Randomly generated datasets

Although the global search is able to obtain solutions with a lower number of services, a first look at the results with the WSC dataset might suggest that the difference of both strategies is

not very significant, as most of the obtained solutions have the same number of services. However, this may be due to a bias in the repository, since all the datasets of the WSC are generated using the same random model. In order to better evaluate and characterize the performance of the hybrid algorithm, we generated a new set of five random datasets that range from 1,000 to 9,000 services. These datasets are available at `https://wiki.citius.usc.es/inv:downloadable_results:ws-random-qos`. Table 5.5 shows the solutions obtained.

Table 5.5: Validation with random datasets

		R-01	**R-02**	**R-03**	**R-04**	**R-05**
#Services in the dataset		1,000	3,000	5,000	7,000	9,000
Validation with Response Time						
Optimal Response Time (ms)		1,430	975	805	1,225	1,420
#Graph Services		54	168	285	383	499
#Graph Services (opt)		22	50	54	56	99
Local Search	#Services	7	18	20	15	19
	Time (s)	0.183	0.403	0.422	0.515	0.641
Global Search	#Services	7	14	15	15	16
	Time (s)	0.243	0.767	4.088	0.740	3.131
Validation with Throughput						
Optimal Throughput (inv/s)		1,000	2,500	1,500	2,000	2,500
#Graph Services		54	168	285	383	499
#Graph Services (opt)		19	46	133	116	103
Local Search	#Services	7	17	24	19	23
	Time (s)	0.072	0.143	0.606	0.732	0.450
Global Search	#Services	7	12	12	15	16
	Time (s)	0.155	0.310	2.479	1.485	1.714

We found that in these datasets, the solutions obtained with the global search strategy are, on average, $\approx 16\%$ smaller than the ones obtained with the local search, whereas the differences in seach time are less pronounced than in the previous experiment. These findings suggest that the performance of each strategy highly depends on the underlying structure of the service repository, which is mostly determined by the number of services and the existing matching relations.

In order to test whether these differences are statistically significant or not, we conducted a nonparametric test using the *binomial sign test* for two dependent samples with a total of 20 datasets (5 WSC w/response time + 5 WSC w/throughput + 5 Random w/response

time + 5 Random w/throughput). The null hypothesis was rejected with *p-value* ≈ 0.01 [94], meaning that both strategies (local and global search) find significantly different solutions. Thus, a hybrid strategy can perform better in many different scenarios, since it achieves a good tradeoff between quality and execution time.

This evaluation shows that, on one hand, the combination of local and global optimization is a general and powerful technique to extract optimal compositions in diverse scenarios, as it brings the best of both worlds. This is specially important when only a little or nothing is known concerning the structure of the underlying repository of services. On the other hand, the results obtained with the Web Service Challenge 2009-2010 show that the hybrid strategy performs better than the state-of-the-art, obtaining solutions with less services and optimal QoS.

5.8 Conclusions

In this paper we have presented a hybrid algorithm to automatically build semantic input-output based compositions minimizing the total number of services while guaranteeing the optimal QoS. The proposed approach combines a set of graph optimizations and a local-global search to extract the optimal composition from the graph. Results obtained with the Web Service Challenge 2009-2010 datasets show that the combination of graph optimizations with a local-global search strategy performs better than the state-of-the-art, as it obtained solutions with less services and optimal QoS. Moreover, the evaluation with a set of randomly generated datasets shows that the hybrid strategy is well suited to perform compositions in diverse scenarios, as it can achieve a good tradeoff between quality and execution time.

5.A Computational Complexity

The calculation of the optimal QoS can be computed in polynomial time for a given *Service Match Graph* using classical shortest path algorithms such as Dijkstra or Bellman-Ford. But, as stated in the introduction, there can exist multiple solutions with the same global QoS but different number of services. Thus, in many scenarios, optimizing the QoS objective function is not enough to provide the best possible answer. However, it turns out that optimizing the number of services of a composition is an intractable problem. The next theorem proves that the Service Minimization Problem (SMP) is a NP-Hard combinatorial optimization problem.

Theorem. *Finding the minimum number of services whose outputs match a given set of unresolved (unmatched) concepts is a NP-Hard combinatorial optimization problem.*

Proof. We will show that the Service Minimization Problem (SMP) is NP-Hard by proving that the optimization version of the Set Cover Problem (SCP), a well-known NP-Hard problem, is polynomial-time *Karp* reducible to SMP $SCP \leq_P SMP$. The optimization version of the SCP problem is defined as follows: given a set of elements $U = \{u_1, \ldots, u_m\}$ and a set S of subsets of U, find the smallest set (cover) $C \subseteq S$ of subsets of S whose union is U. The decision version of this problem, stated as that of deciding whether exists a cover C_{SCP} of size k or less ($|C_{SCP}| \leq k$), is NP-Complete. We will also consider the simplest form of the SMP that can be contained in a *Service Match Graph*, which is defined as follows: given a service w_U and a set of candidate services $W_S = \{w_1, \ldots, w_n\}$ such that $O_{w_1} \otimes I_{w_U} \neq \emptyset \wedge \cdots \wedge O_{w_n} \otimes I_{w_U} \neq \emptyset$, select the smallest subset of services from W_S such that the union of the outputs of the services from W_S, O_{W_S}, satisfies $O_{W_S} \otimes I_{w_U} = I_{w_U}$, i.e., the outputs of the services contained in W_S match all the inputs of w_U. As in the SCP, the decision version of this optimization problem is defined as that of deciding whether exists a subset of candidate services C_{SMP} of size k or less ($|C_{SMP}| \leq k$) such that the union of the outputs of the services in C_{SMP} match all the inputs of w_U.

In order to prove that the SMP optimization problem is NP-Hard, we need to demonstrate that its corresponding decision problem is NP-Complete. We will therefore reduce the SCP problem by means of a function φ that transforms any arbitrary instance of the SCP into an instance of the SMP in polynomial time. We have to prove that 1) $\varphi(U,S)$ is a SMP problem; 2) φ runs in polynomial time; and 3) there is a set covering of $\varphi(U,S)$ of size k or less if and only if there is a set covering of U in S of size k or less.

Given a pair (U,S), we define $\varphi(U,S) = (w_U, W_S)$ such that:

- $w_U = \{I_{w_U} = U = \{u_1, \ldots, u_n\}, \emptyset\}$, where u_i is the ith unresolved input of w_U.
- $\forall s_i = \{u_{i_1}, \ldots, u_{i_n}\} \in S$, $\exists w_i \in W_S$ such that $w_i = \{\emptyset, O_{w_i}\}$ and $O_{w_i} \otimes I_{w_U} = s_i$

By this definition, the $\varphi(U,S)$ maps each element $u \in U$ to an input of the service w_U. Each subset $s_i \in S$ is also mapped to a service whose outputs match exactly the inputs of w_U that correspond with the elements of s_i. This mapping can be computed by adding a match from an arbitrary output of each service $w_i \in W_S$ to each input $u_i \in s_i$, which clearly runs in

linear time in the size of U. Moreover, $\varphi(U,S)$ is a Service Minimization Problem according to its definition.

Now suppose there is a set covering $|C| \leq k, C \subseteq S$ of U. Thus, $\forall u \in U, \exists c_i \in C$ such that $u \in c$. From the services (w_U, W_S) constructed from (U,S) by $\varphi(U,S)$, there exists $w_i \in W_S$ such that $O_{w_i} \otimes I_{w_U} = c_i \subseteq I_{w_U}$, and so $\bigcup_i (O_{w_i} \otimes I_{w_U}) = I_{w_U} = C$, i.e., the outputs of the services from the set W_S of size k or less represent a cover of the Service Minimization Problem $\varphi(U,S)$.

□

5.B Algorithm Analysis and Discussion

The proposed approach consists of a hybrid algorithm that optimizes both the global QoS and selects the composition with the minimum number of services that preserves the optimal QoS. As demonstrated in Appendix 5.A, the problem of minimizing the number of services is NP-Hard. Thus, under the $P \neq NP$ assumption, there is no polynomial time algorithm that can exactly solve this optimization problem. However, although it is in general intractable, in practice many instances of the problem, as shown in the evaluation section, can be optimally solved in reasonable time. In those situations, it may be preferable to provide optimal solutions instead of just sub-optimal ones. Our approach takes advantage of a hybrid strategy that combines a local search and a global search plus the use of preprocessing optimizations and search optimizations (minimum-remaining-values heuristic, cycle detection, QoS bounds propagation) in order to achieve a good trade-off between optimality of the solution and computation time. Here we analyze the complexity of the proposed techniques.

5.B.1 Cycle detection

The cycle detection is implemented as a Look-Ahead strategy, that traverses all the resolved matches, starting from the current service (the one selected to resolve a new unresolved input), until no more services are reachable. This strategy seeks to discover whether the current service is a valid candidate or not by checking if it can lead to a dependency cycle, so it can be prematurely discarded. The cycle detection algorithm takes $O(|V| + |E|)$, since every service, input, output and match between inputs and outputs have to be traversed in worst-case.

5.B.2 QoS Update

The QoS update method calculates the optimal end-to-end QoS through the graph. This method is also used to recalculate optimal QoS bounds whenever a local QoS bound is exceeded. This problem can be modeled as a shortest path problem with generalized costs for QoS (as shown in Section 4.3) and solved using Dijkstra's algorithm. The worst-case time complexity of this algorithm is as follows: given a *Service Match Graph* $G_S = (V,E)$, where $W_R \subset V$ is the set of services in the graph, there are at most $|W_R|$ calls to *POP* method to extract the lowest scored service from the queue. Since the queue is implemented as a binary heap, the *POP* and *INSERT* methods have a time of $O(log(n))$, where n is the size of the queue. Thus, in the worst case, the running time is $O(|W_R| \cdot log(|W_R|))$, plus the (at most) $|E|$ updates of neighbor services that are reinserted into the queue. Therefore, the overall time is $O((|E| + |W_R|) \cdot log(W_R))$.

5.B.3 Local search

This method performs a heuristically guided local search to minimize the number of services of the optimal end-to-end QoS composition. At each step, it selects the most promising candidate by selecting the one with fewer inputs that matches the largest number of unresolved inputs. If the algorithm gets stuck at some point, i.e., it reaches a point where no service can be selected without leading to a cyclic dependency, it backtracks to try the next most promising candidate service. The algorithm calls *RANK-RESOLVERS* to rank the candidates according to the number of unresolved inputs that each candidate can match and, in case of draw the service with less inputs is preferred. The sorting of services takes $O(n \cdot log(n))$ using merge sort, where n is the number of services. Each time a service is selected, the method *RESOLVE* creates an updated copy of the graph in $O(|V| + |E|)$.

Assumming non-cyclic dependencies in the *Service Match Graph*, in the worst case the algorithm have to select all the services from the graph until no unresolved inputs are left. Thus, in the first step $t_{|W_R|}$ the algorithm ranks all the $|W_R|$ services in $O(|W_R| \cdot log(|W_R|))$, selects the first one and generates a new copy of the graph in $O(|V| + |E|)$. The running time of this step is $O(|W_R| \cdot log(|W_R|) + O(|V| + |E|) = O(|W_R| \cdot log(|W_R|))$. In the next step $t_{|W_R|-1}$, the algorithm ranks $|W_R| - 1$ services, selects the best one, creates a copy of the graph and so on. Therefore, the asymptotic upper bound of the running time of $t_{|W_R|} + t_{|W_R|-1} + \cdots + t_1$ is $O(|W_R| \cdot log(|W_R|))$.

In the absence of the assumption of non-cyclic dependencies, the asymptotic upper bound analysis shows that the time complexity grows exponentially with the depth of the search, since in the worst-case the algorithm fails (backtracks) at each step until the last combination of services is explored. However, in practice, this upper bound seems far from the average-case. As shown in the evaluation (Section 6), the growth of the time with respect to the size of the graph is closer to the best-case scenario, since an exponential number of backtracks due to cylic dependencies is extremely rare. In any case, the algorithm can be easily adapted to perform better in the worst-case scenario, for example by limiting the number of candidates to the top-K best services for each unresolved input.

5.B.4 Global search

The aim of the global search algorithm is to perform an exhaustive search to find the minimum combination of services that satisfy the composition request with optimal QoS. The algorithm explores every possible valid combination of services in a breadth-first fashion by resolving one input at a time. For each unresolved input with $k > 1$ candidates, new k different states are created by calling the *RESOLVE* method and pushed to the queue for further expansion. In order to calculate an asymptotic upper bound for the time complexity, we can compute the number of combinations of services that the algorithm needs to extract from the queue in the worst-case. To this end, we first count the maximum number of combinations (solutions) that we can generate for a simple graph with fixed size and then we generalize the problem for a graph of any size.

Left graph from Figure 5.3 shows an example of a *Service Match Graph* with 4 services (excluding *Si* and *So*). As can be seen, *Si* requires two inputs, 1 and 2. On the other hand, the outputs of A and B match the input 1 whereas the outputs of services C and D match the input 2. Therefore, in order to match both inputs, we can select services A and C, A and D, B and C or B and D (2×2 combinations). By computing all possible combinations, we can reduce the graph from the left, where *Si* has two inputs, to the graph from the right, where *Si* has just one input.

In general, given a service w with $|I_w| = k$ inputs and $c_1, c_2, \ldots, c_k$ set of candidate services for each input, there are $\prod_i |c_i|$ combinations of services, i.e., we can replace the k inputs with k sets of candidate services by one input with $\prod_i |c_i|$ candidates. Since each service can have in turn some inputs with other candidates, we can recursively replace each service with all the possible combinations of services that can be generated. This process leads to

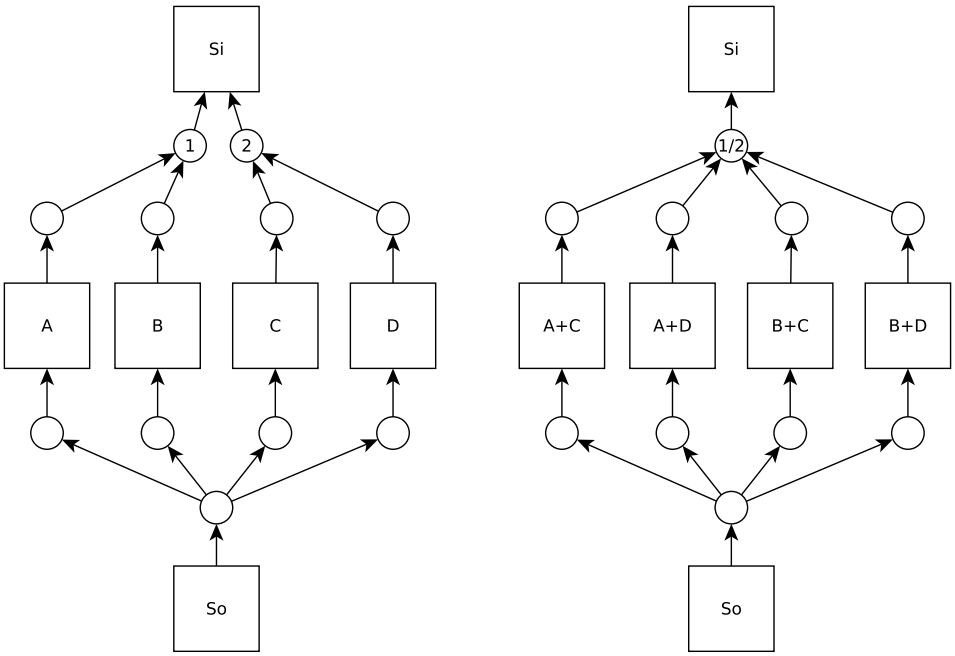

Figure 5.3: Reduction of the left graph into the right graph by computing all possible combinations of services

a flattening of the graph until there is just one level with all the possible combinations of services (compositions) that can be generated for a given *Service Match Graph*. Thus, the problem of counting the number of possible solutions in the worst-case can be reduced to the following: given a *Service Match Graph* with $|W_R|$ services, what is the maximum of products of partitions of W_R? More formally, given a set S ($|S| \geq 1$), choose n partitions $c_1, c_2, \ldots, c_n$ such that $\sum_i |c_i| = |S|$ and $\prod_i |c_i|$ is maximized. For example, given 11 services, we can take 3 groups of 3 services and one with the remaining 2 services, so the product of the partition is $3^3 \cdot 2 = 54$, which is the maximum. Finding an upper bound for this value will gives us an upper bound for the maximum number of compositions that can be enumerated in the worst-case, i.e., for the most complex *Service Match Graph* that can be generated with $|W_R|$ services. It can be proved that, for any set of size n, the maximum can be obtained by partitioning the set into groups of 2 and 3 elements, with no more than 2 groups of 2 elements. From this it

follows that the maximum product is bounded by $3^{n/3}$, so we can conclude that $O(3^{n/3})$ is a tight asymptotic upper bound on the running time in the worst-case.

However, it should be noted that although the calculation of an optimal solution for the problem in the worst-case requires exponential time with the size of the graph, in practice, the number of services for a particular request is usually orders of magnitude lower that the number of available services in the dataset (see Table 2 and 4). In addition to this, the optimizations introduced in Section 5.3 plus the global QoS bound propagation, the minimum-remaining-values heuristic and cycle detection used in the global search are aimed to reduce further the size of the explored search space by decreasing the number of analyzed services.

Chapter 6

Conclusions

The growing importance of Service Oriented Computing (SOC) within the domain of distributed computing has led to an important increase in the number of available Web services both inside and outside of different organizations and companies worldwide. One major advantage of Web services is the possibility to combine them to create composite services on-demand by means of automatic composition techniques. When a single web service cannot meet complex business requirements, different services can be combined together to build composite web services that fulfill a request. As the number of available services on the Internet grows, the need to develop efficient and optimal algorithms that can deal with a large number of services taking into account the QoS becomes a challenging task.

In this thesis we presented a set of techniques to generate optimal and fast automatic compositions of Web services based on the input-output matching of services' interfaces. We started with an exploratory research focusing on the use of control-centric techniques to generate expressive composition workflows by combining different control structures. For this purpose, we developed a Genetic Programming algorithm that uses a context-free grammar and a set of genetic operators and optimizations to generate valid composition workflows. Although the results obtained with this technique demonstrate the effectiveness of this approach to generate expressive solutions exploiting the different control structures defined in the formal grammar, the complexity of the search space and the elevated computation time required to generate good solutions makes this technique innapropiate for generating compositions on the fly for some large instances of the problem, but appropiate as a powerful tool for offline optimization of composition workflows. In order to cope with this limitation, we moved to-

wards a data-centric approach based on the idea of building service dependency graphs (or service match graphs) by analyzing the semantic information of the inputs and outputs of services. Using the information of a particular composition request, we proposed a way to construct an optimized graph with all the candidate services and their I/O matches as well as different algorithms to extract optimal compositions from the graph. We also proposed and studied the integration of the service discovery and service composition via a fine-grained I/O used to discover relevant services during the generation of the graph. All these ideas have been integrated into a graph-based framework for service composition, and its effectiveness has been studied by means of an open-source reference implementation of the framework, using different optimization mechanisms to minimize the overhead of the service discovery.

The main conclusions are summarized as follows:

1. In Chapter 2 a genetic programming algorithm for web services composition has been presented. Given an input-output based composition request, the algorithm is able to generate a complete optimized composition workflow from scratch, using different control structures defined in a context-free grammar, and optimizing the number of services and the runpath of the solutions. A full validation has been done for eight different composition problems coming from four different repositories with 158, 558, 1,000, and 1,090 services. The results showed that the proposed approach can be used to effectively generate correct workflow-based compositions using a wide variety of control constructions. The algorithm can be used to generate solutions on the fly for medium-sized repositories, as it obtained correct solutions in less than a second. However, for larger repositories, the amount of computation time required to generate correct solutions becomes prohibitive to be used at runtime, making this technique better suited for offline optimization of workflows.

2. In Chapter 3, we presented a graph-based algorithm for automatic composition based on a heuristic search method to extract optimal solutions from a graph of services. We introduced different optimization techniques to reduce the complexity of the graph and to detect and prune redundant nodes during the search. The validation with the eight repositories from Web Service Challenge 2008 showed that the algorithm is capable of obtaining optimal solutions in all datasets, improving the results of the state-of-the-art.

3. In Chapter 4 we presented a theoretical analysis of service composition in terms of its dependency with service discovery and we defined a formal integrated graph-based

composition framework that incorporates the service discovery task as a central activity in the composition process. We also devised a reference implementation of this framework on the basis of two open-source components, namely iServe and ComposIT. The reference implementation has been used to empirically study the impact of discovery and matchmaking on service composition under different conditions. Our empirical analysis shows that: 1) typical approaches followed by discovery engines cannot serve as a suitable basis to support efficient service composition as they lead to prohibitive execution times, and 2) with the adequate interface granularity and indexing, discovery engines can support highly efficient composition akin to that obtained by the fastest composition engines without having to assume to local availability and in-memory preloading of service registries.

4. In Chapter 5 we presented an extension of the graph-based framework to incorporate QoS optimization. The extension includes new graph optimizations that take into account the QoS of the services and a hybrid algorithm that combines a local search and a global search to achieve a good tradeoff between computation time and optimality. Results obtained with the Web Service Challenge 2009-2010 datasets showed that the combination of graph optimizations with a local-global search strategy performed better than the state-of-the-art, as it obtained solutions with less services and optimal end-to-end QoS.

As a future work, there is a number of research directions that we think are worth to explore to further improve the work presented in this dissertation:

- *Improvement of the hybrid algorithm by reusing information of the local search into the global search*. The information gathered by the local search algorithm can be used to collect useful information about the structure of the search space. This information can be used to detect dead-ends or sub-optimal paths that can be pruned lately by the global search algorithm.

- *Support for complex end-to-end QoS constraints*. Users may prefer solutions that are good enough for their needs rather than just optimal solutions. These solutions can be expressed by enforcing global constraints on the quality attributes of the composition. For example, an user may be consider a good solution one that meets $responseTime \leq 100$ AND $throughput > 1500$. Simple QoS constraints can be currently implemented

adjusting the QoS bounds used by the hybrid algorithm. These bounds can be used to prune those services (or combinations of services) that exceed a certain threshold. More complex QoS constraints involving many simultaneous QoS attributes would require the use of more advanced techniques typically used to solve Constraint Satisfaction Problems (CSPs). However, since the proposed framework is graph-based, any kind of graph-based CSP method can be naturally introduced to allow more advanced constraint handling.

- *Multiobjective optimization*. One of the main problems in service composition is how to generate optimal compositions when each service is associated with many different QoS attributes. One strategy that can be adopted is to combine all the attributes into a single objective function (or utility function) through a linear combination of the weighted attributes. We have already demonstrated that this approach can be used to extract good suboptimal solutions from the composition graph [97]. However, there are some limitations in this approach. On one hand, the use of custom weights to modulate the importance of each QoS attribute is hard to adjust by hand, since 1) weights are too fine-grained for an user to model the importance of each attribute and 2) small changes in the weights can drastically affect the overall quality of the solutions that the algorithm can obtain. On the other hand, there is usually no single optimal solution but many different non-dominated solutions that are better in some attributes but worse in others. An interesting alternative is to integrate multiobjective optimization algorithms to generate the full non-dominated Pareto set of solutions and let the user pick the one that better fits his needs.

- *Support of world-altering services with explicit preconditions and effects*. The current composition framework does not include preconditions and effects as it is focused on information-providing services, i.e., it only uses the information of inputs and outputs to create compositions. This limits the type of compositions that can be generated. For example, a `SellerService` used to sell items and process electronic payments is a type of a world-altering service that may require some *preconditions* related to the state of the world for its invocation (a valid credit card with enough money) and also may produce some *effects* that alter the current state of the world (the credit card is charged and the number of available items changes). We plan to extend the current framework to support also the composition of world-altering services with explicit preconditions and effects.

Bibliography

[1] Vikas Agarwal, Girish Chafle, Sumit Mittal, and Biplav Srivastava. Understanding approaches for web service composition and execution. In *Proceedings of the 1st Bangalore Annual Compute Conference, Compute 2008*, page 1, 2008.

[2] Rohit Aggarwal, Kunal Verma, John A. Miller, and William Milnor. Constraint driven web service composition in METEOR-S. In *IEEE International Conference on Services Computing (SCC 2004)*, pages 23–30, 2004.

[3] Marco Aiello, Nico van Benthem, and Elie el Khoury. Visualizing Compositions of Services from Large Repositories. In *Proceedings of the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services (EEE'08)*, pages 359–362, 2008.

[4] Marco Aiello, Elie el Khoury, Alexander Lazovik, and Patrick Ratelband. Optimal qos-aware web service composition. In *IEEE Conference on Commerce and Enterprise Computing, CEC 2009*, pages 491–494, 2009.

[5] Rama Akkiraju, Biplav Srivastava, Anca Andreea Ivan, Richard Goodwin, and Tanveer Syeda-Mahmood. SEMAPLAN: Combining planning with semantic matching to achieve Web service composition. In *IEEE International Conference on Web Services (ICWS 2006)*, pages 37–44, 2006.

[6] Atif Alamri, Mohamad Eid, and Abdulmotaleb El Saddik. Classification of the state of the art dynamic web services composition techniques. *International Journal of Web and Grid Services*, 2(2):148–166, 2006.

[7] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: concepts, architectures and applications*. Data-Centric Systems and Applications. Springer, October 2004.

[8] Mohammad Alrifai and Thomas Risse. Combining global optimization with local selection for efficient QoS-aware service composition. In *18th International Conference on World Wide Web (WWW '09)*, pages 881–890, 2009.

[9] Bonnie Brinton Anderson, James V. Hansen, and Paul Benjamin Lowry. Creating automated plans for Semantic Web applications through planning as model checking. *Expert Systems with Applications*, 36(7):10595–10603, 2009.

[10] Ali Arsanjani. Service-Oriented Modeling and Architecture.

[11] L. Aversano, G. Canfora, and A. Ciampi. An algorithm for web service discovery through their composition. In *IEEE International Conference on Web Services (ICWS 2004)*, pages 332–339, 2004.

[12] Lerina Aversano, Massimiliano di Penta, and Kunal Taneja. A genetic programming approach to support the design of service compositions. *International Journal of Computer Systems Science and Engineering*, 4:247–254, 2006.

[13] Ajay Bansal, M. Brian Blake, Srividya Kona, Steffen Bleul, Thomas Weise, and Michael C. Jaeger. WSC-08: Continuing the Web Services Challenge. In *10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services*, pages 351–354, 2008.

[14] Lina Barakat, Simon Miles, Iman Poernomo, and Michael Luck. Efficient Multi-Granularity Service Composition. In *IEEE International Conference on Web Services (ICWS)*, pages 227–234, 2011.

[15] Peter Bartalos and Mária Bieliková. Automatic Dynamic Web Service Composition: A Survey and Problem Formalization. *Computing and Informatics*, 30:793–827, 2011.

[16] Rainer Berbner, Michael Spahn, Nicolas Repp, Oliver Heckmann, and Ralf Steinmetz. Heuristics for QoS-aware Web Service Composition. In *International Conference on Web Services (ICWS'06)*, pages 72–82, 2006.

[17] Piergiorgio Bertoli, Marco Pistore, and Paolo Traverso. Automated composition of Web services via planning in asynchronous domains. *Artificial Intelligence*, 174(3-4):316–361, 2010.

[18] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[19] Antonio Brogi, Sara Corfini, and Razvan Popescu. Semantics-based composition-oriented discovery of web services. *ACM Transactions on Internet Technology*, 8(4), 2008.

[20] Mark Burstein, Jerry Hobbs, Ora Lassila, Drew Mcdermott, Sheila Mcilraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. Owl-s: Semantic markup for web services. `http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/`, November 2004. Online; accessed May-2015.

[21] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for QoS-aware service composition based on genetic algorithms. In *Genetic and Evolutionary Computation Conference, GECCO 2005*, pages 1069–1075, 2005.

[22] Jorge Cardoso, Amit Sheth, John Miller, Jonathan Arnold, and Krys Kochut. Quality of service for workflows and web service processes. *Web Semantics*, 1:281–308, 2004.

[23] Jorge Cardoso and Amit P. Sheth. Semantic e-workflow composition. *Journal of Intelligent Information Systems*, 21(3):191–225, 2003.

[24] Mark Carman, Luciano Serafini, and Paolo Traverso. Web Service Composition as Planning. In *Workshop on planning for web services (ICAPS 2003)*, pages 1636–1642, 2003.

[25] Bernard Carré. *Graphs and networks*. Oxford University Press, 1979.

[26] Wei-Chun Chang, Ching-Seh Wu, and Chun Chang. Optimizing Dynamic Web Service Component Composition by Using Evolutionary Algorithms. In *IEEE / WIC / ACM International Conference on Web Intelligence (WI 2005)*, pages 708–711, 2005.

[27] Kun Chen, Jiuyun Xu, and Stephan Reiff-Marganiec. Markov-htn planning approach to enhance flexibility of automatic web service composition. In *IEEE International Conference on Web Services (ICWS 2009)*, pages 9–16, 2009.

[28] Min Chen and Yuhong Yan. Redundant Service Removal in QoS-Aware Service Composition. *IEEE International Conference on Web Services (ICWS)*, 2012.

[29] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, International Business, and Machines Corporation. Web Services Description Language (WSDL) 1.1, 2001.

[30] Dongjie Chu, Jun Han, Jing Li, and Yongwang Zhao. XSSD: A Fast Hybrid Semantic Web Services Discovery Method. In *International Conference on Computer Technology and Development, 3rd (ICCTD 2011)*, 2011.

[31] Marco Comuzzi and Barbara Pernici. An architecture for flexible web service qos negotiation. In *Ninth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2005)*, pages 70–82, 2005.

[32] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. *Business Process Execution Language for Web Services, Version 1.0*, November 2002.

[33] Francisco Curbera, William A. Nagy, and Sanjiva Weerawana. Web Service: Why and How. In *Proceedings of the OOPSLA-2001 Workshop on Object-Oriented Services*, Tampa, Florida, USA, 2001.

[34] Eduardo Gonçalves da Silva, Luís Ferreira Pires, and Marten van Sinderen. Towards runtime discovery, selection and composition of semantic services. *Computer Communications*, 34(2):159–168, February 2011.

[35] Andrea D'Ambrogio. A model-driven WSDL extension for describing the qos ofweb services. In *IEEE International Conference on Web Services (ICWS 2006)*, pages 789–796, 2006.

[36] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[37] Dieter Fensel. *Ontologies*. Springer, 2001.

[38] Toktam Ghafarian and Mohsen Kahani. Semantic web service composition based on ant colony optimization method. *First International Conference on Networked Digital Technologies*, pages 171–176, 2009.

[39] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning: theory & practice*. Elsevier, 2004.

[40] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions of Systems Science and Cybernetics*, 4(2):100–107, 1968.

[41] Seyyed Hashemian and Farhad Mavaddat. A Graph-Based Framework for Composition of Stateless Web Services. In *European Conference on Web Services (ECOWS'06)*, pages 75–86, 2006.

[42] Seyyed Vahid Hashemian and Farhad Mavaddat. A graph-based approach to web services composition. In *IEEE/IPSJ International Symposium on Applications and the Internet (SAINT 2005)*, pages 183–189, 2005.

[43] Ourania Hatzi, Dimitris Vrakas, Mara Nikolaidou, Nick Bassiliades, Dimosthenis Anagnostopoulos, and Ioannis Vlahavas. An Integrated Approach to Automated Semantic Web Service Composition through Planning. *IEEE Transactions on Services Computing*, 5(3):1–14, 2012.

[44] Patrick Hennig and Wolf-Tilo Balke. Highly Scalable Web Service Composition Using Binary Tree-Based Parallelization. *IEEE International Conference on Web Services*, pages 123–130, 2010.

[45] Jörg Hoffmann, Piergiorgio Bertoli, and Marco Pistore. Web Service Composition as Planning, Revisited: In Between Background Theories and Initial State Uncertainty. In *Proceedings of the 22nd national conference on Artificial intelligence (AAAI'07)*, pages 1013–1018. AAAI Press, 2007.

[46] Wei Jiang, Songlin Hu, Zhenqiu Huang, Zhiyong Liu, and Qos Data Handler. Two-Phase Graph Search Algorithm for QoS-Aware Automatic Service Composition. In *International Conference on Service-Oriented Computing and Applications*, pages 1–4, 2010.

[47] Wei Jiang, Songlin Hu, and Zhiyong Liu. Top K Query for QoS-Aware Automatic Service Composition. volume 7, pages 681–695, October 2014.

[48] Wei Jiang, Charles Zhang, Zhenqiu Huang, Mingwen Chen, Songlin Hu, and Zhiyong Liu. QSynth: A Tool for QoS-aware Automatic Service Composition. In *IEEE International Conference on Web Services*, pages 42–49, 2010.

[49] Matthias Klusch. Overview of the S3 Contest: Performance Evaluation of Semantic Service Matchmakers. In *Semantic Web Services*, pages 17–34. Springer, 2012.

[50] Matthias Klusch and Andreas Gerber. Fast Composition Planning of OWL-S Services and Application. In *Proceedings of the European Conference on Web Services (ECOWS'06)*, pages 181–190, 2006.

[51] Matthias Klusch, Andreas Gerber, and Marcus Schmidt. Semantic Web Service Composition Planning with OWLS-Xplan. In *Proceedings of the AAAI Fall Symposium on Semantic Web and Agents*, 2005.

[52] Matthias Klusch and Patrick Kapahnke. OWLS-MX3: an adaptive hybrid semantic service matchmaker for OWL-S. In *Proceedings of 3rd Int. Workshop on Semantic Matchmaking and Resource Retrieval*, 2009.

[53] Srividya Kona, Ajay Bansal, M Brian Blake, Steffen Bleul, and Thomas Weise. WSC-2009: a quality of service-oriented web services challenge. In *IEEE International Conference on Commerce and Enterprise Computing*, pages 487–490, 2009.

[54] Srividya Kona, Ajay Bansal, M. Brian Blake, and Gopal Gupta. Generalized Semantics-Based Service Composition. In *IEEE International Conference on Web Services (ICWS)*, pages 219–227. IEEE, 2008.

[55] Jacek Kopecký, Tomas Vitvar, Carine Bournez, and Joel Farrell. SAWSDL: semantic annotations for WSDL and XML schema. *IEEE Internet Computing*, 11(6):60–67, 2007.

[56] Ulrich Kuster, Birgitta Konig-Ries, and Andreas Krug. OPOSSum - An Online Portal to Collect and Share SWS Descriptions. In *Proceedings of the 2th IEEE International Conference on Semantic Computing (ICSC 2008)*, pages 480–481, Santa Clara, California, USA, 2008.

[57] Freddy Lécué, Eduardo Silva, Luis Ferreira Pires, and F Lecue. A framework for dynamic web services composition. *2nd ECOWS Workshop on Emerging Web Services Technology*, pages 59–75, 2007.

[58] Q.A. Liang and S.Y.W. Su. AND/OR Graph and Search Algorithm for Discovering Composite Web Services. *International Journal of Web Services Research*, 2(4):48–67, 2005.

[59] M. Klusch and P. Kapahnke. iSem: Approximated reasoning for adaptive hybrid selection of semantic services. In *The semantic web: Research and applications*, pages 30–44. Springer, 2010.

[60] Therani Madhusudan and Naveen Uttamsingh. A declarative approach to composing web services in dynamic environments. *Decision Support Systems*, 41(2):325–357, 2006.

[61] George Markou and Ioannis Refanidis. Non-deterministic planning methods for automated web service composition. *Artificial Intelligence Research*, 5(1):14, 2016.

[62] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. *OWL-S: Semantic Markup for Web Services*. World Wide Web Consortium (W3C), November 2004.

[63] Sheila A. McIlraith and Tran Cao Son. Adapting Golog for Composition of Semantic Web Services. In *Proceedings of the Eighth International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, pages 482–496, 2002.

[64] Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng. Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

[65] Brahim Medjahed, Athman Bouguettaya, and Ahmed K. Elmagarmid. Composing web services on the semantic web. *The International Journal on Very Large Data Bases*, 12(4):333–351, 2003.

[66] Nikola Milanovic and Miroslaw Malek. Search Strategies for Automatic Web Service Composition. *International Journal of Web Services Research*, 3(2):1–32, 2006.

[67] Johan Montagnat, Tristan Glatard, and Diane Lingrand. Data composition patterns in service-based workflows. In *Workshop on Workflows in Support of Large-Scale Science*, pages 1–10. IEEE, 2006.

[68] Wonhong Nam, Hyunyoung Kil, and Dongwon Lee. Type-Aware Web Service Composition Using Boolean Satisfiability Solver. *10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services*, 1:331–334, 2008.

[69] Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, William Murdock, Dan Wu, and Fusun Yaman. SHOP2: an HTN planning system. *Journal of Artificial Intelligence Research*, 20(4):379–404, 2003.

[70] S.C. Oh, D. Lee, and S.R.T. Kumara. Web service planner (WSPR): an effective and scalable web service composition algorithm. *International Journal of Web Services Research (IJWSR)*, 4(1):1–22, 2007.

[71] Seog-Chan Oh, Hyunyoung Kil, Dongwon Lee, and Soundar R. T. Kumara. WSBen: A Web Services Discovery and Composition Benchmark. In *IEEE International Conference on Web Services (ICWS 2006)*, pages 239–248, 2006.

[72] Seog-Chan Oh, Dongwon Lee, and Soundar R.T. Kumara. A comparative illustration of AI planning-based web services composition. *ACM SIGecom Exchanges*, 5(5):1–10, January 2006.

[73] Seog-Chan Oh, Dongwon Lee, and Soundar R.T. Kumara. Effective Web Service Composition in Diverse and Large-Scale Service Networks. *IEEE Transactions on Services Computing*, 1(1):15–32, 2008.

[74] Seog-Chan Oh, Ju-Yeon Lee, Seon-Hwa Cheong, Soo-Min Lim, Min-Woo Kim, Sang-Seok Lee, Jin-Bum Park, Sang-Do Noh, and Mye M. Sohn. WSPR*: Web-Service Planner Augmented with A* Algorithm. In *IEEE Conference on Commerce and Enterprise Computing*, 2009.

[75] B. On and E.J. Larson. BF*: Web Services Discovery and Composition as Graph Search Problem. *IEEE International Conference on e-Technology, e-Commerce and e-Service*, (1):784–786, 2005.

[76] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. Semantic Matching of Web Services Capabilities. In *The Semantic Web - ISWC 2002*, pages 333–347, 2002.

[77] Michael P Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *Computer*, 40(11):38–45, 2007.

[78] Mike P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *4th International Conference on Web Information Systems Engineering, WISE 2003, Rome, Italy, December 10-12, 2003*, pages 3–12, 2003.

[79] Mike P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-Oriented Computing: a Research Roadmap. *International Journal of Cooperative Information Systems*, 17(2):223–255, 2008.

[80] M.P. Papazoglou and D. Georgakopoulos. Service-Oriented Computing. *Communications of the ACM*, 46(70):25–28, 2003.

[81] C. Pedrinaci, D. Liu, M. Maleshkova, D. Lambert, J. Kopecky, and J. Domingue. iServe: a linked services publishing platform. In *CEUR Workshop Proceedings*, volume 596, 2010.

[82] Carlos Pedrinaci and John Domingue. Toward the Next Wave of Services: Linked Services for the Web of Data. *Journal of Universal Computer Science*, 16(13):1694–1719, 2010.

[83] Carlos Pedrinaci, John Domingue, and Amit P Sheth. *Semantic Web Services*, pages 977–1035. Springer, 2011.

[84] Joachim Peer. Web Service Composition as AI Planning – a Survey. Technical Report March, University of St. Gallen, Switzerland, 2005.

[85] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, P. Traverso, Christoph Bussler, and Dieter Fensel. Planning and monitoring web service composition. *Artificial Intelligence: Methodology, systems and applications*, 3192:106–115, 2004.

[86] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated composition of web services by planning at the knowledge level. In *Proceedings of the 19th international joint conference on Artificial intelligence (IJCAI'05)*, pages 1252–1259, 2005.

[87] Shankar R Ponnekanti and Armando Fox. SWORD : A Developer Toolkit for Web Service Composition. In *11th World Wide Web Conference*, 2002.

[88] Lukasz Radziwonowicz, Daniel Schreckling, and Marko Vujasinovic. D31.3.2 - Assisted Service Composition Engine – Final prototype. Public deliverable, The COMPOSE Project (FP7-ICT 317862), 2015.

[89] K. Raman, Yue Zhang Yue Zhang, M. Panahi, and Kwei-Jay Lin Kwei-Jay Lin. Customizable Business Process Composition with Query Optimization. *2008 10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conferebce on Enterprise Computing, E-Commerce and E-Services*, 2008.

[90] Shuping Ran. A model for web services discovery with QoS. *SIGecom Exchanges*, 4(1):1–10, 2003.

[91] Jinghai Rao, Peep Küngas, and Mihhail Matskin. Composition of semantic web services using linear logic theorem proving. *Information Systems*, 31(4):340–360, 2006.

[92] Jinghai Rao and Xiaomeng Su. A Survey of Automated Web Service Composition Methods. In *International Workshop on Semantic Web Services and Web Process Composition*, pages 43–54, 2004.

[93] Kaijun Ren, Xiao Liu, Jinjun Chen, Nong Xiao, Junqiang Song, and Weimin Zhang. A QSQL-based Efficient Planning Algorithm for Fully-automated Service Composition in Dynamic Service Environments. In *Proceedings of the 2008 IEEE International Conference on Services Computing (SCC'08)*, pages 301–308, Washington, DC, USA, 2008. IEEE Computer Society.

[94] Ismael Rodríguez-Fdez, Adrián Canosa, Manuel Mucientes, and Alberto Bugarín. STAC: a web platform for the comparison of algorithms using statistical tests. In *Proceedings of the 2015 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, 2015.

[95] Pablo Rodriguez-Mier, Adrian Gonzalez-Sieira, Manuel Mucientes, Manuel Lama, and Alberto Bugarin. Hipster: An Open Source Java Library for Heuristic Search. In *9th Iberian Conference on Information Systems and Technologies*, 2014.

[96] Pablo Rodriguez-Mier, Manuel Mucientes, and Manuel Lama. Automatic Web Service Composition with a Heuristic-Based Search Algorithm. In *IEEE International Conference on Web Services (ICWS 2011)*, pages 81–88, 2011.

[97] Pablo Rodriguez-Mier, Manuel Mucientes, and Manuel Lama. A Dynamic QoS-Aware Semantic Web Service Composition Algorithm. In *International Conference on Service-Oriented Computing*, pages 623–630, 2012.

[98] Pablo Rodríguez-Mier, Manuel Mucientes, Manuel Lama, and Miguel I. Couto. Composition of web services through genetic programming. *Evolutionary Intelligence*, 3(3-4):171–186, 2010.

[99] Pablo Rodríguez-Mier, Manuel Mucientes, Juan Carlos Vidal, and Manuel Lama. An Optimal and Complete Algorithm for Automatic Web Service Composition. *International Journal of Web Service Research*, 9(2):1–20, 2012.

[100] Pablo Rodriguez Mier, Carlos Pedrinaci, Manuel Lama, and Manuel Mucientes. An Integrated Semantic Web Service Discovery and Composition Framework. *IEEE Transactions on Services Computing*, 2015 (DOI 10.1109/TSC.2015.2402679).

[101] Dumitru Roman, Uwe Keller, Holger Lausen, Jos De Bruijn, Michael Stollberg, Axel Polleres, Cristina Feier, Cristoph Bussler, and Dieter Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):76–106, 2005.

[102] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Cristoph Bussler, and Dieter Fensel. Web service modeling ontology. *Applied ontology*, 1(1):77–106, 2005.

[103] M. Rouached, W. Fdhila, and C. Godart. Web Services Compositions Modelling and Choreographies Analysis. *International Journal of Web Services Research (IJWSR)*, 7(2):87–110, 2010.

[104] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2009.

[105] Hadi Saboohi and Sameem Abdul Kareem. A resemblance study of test collections for world-altering semantic web services. In *Proceedings of the International Multi-Conference of Engineers and Computer Scientists (IMECS)*, volume 1, pages 716–720, 2011.

[106] Mohamed Adel Serhani, Rachida Dssouli, Abdelhakim Hafid, and Houari A. Sahraoui. A qos broker based architecture for efficient web services selection. In *IEEE International Conference on Web Services (ICWS 2005)*, pages 113–120, 2005.

[107] Mazen Shiaa, Jan Fladmark, and Benoit Thiell. An Incremental Graph-based Approach to Automatic Service Composition. In *2008 IEEE International Conference on Services Computing*, pages 397–404, 2008.

[108] Evren Sirin and Bijan Parsia. Planning for Semantic Web Services. In *Semantic Web Services Workshop at 3rd International Semantic Web Conference*, pages 33–40, 2004.

[109] Evren Sirin, Bijan Parsia, and James Hendler. Template-based composition of semantic web services. In *In AAAI Fall Symposium on agents and the semantic web*, pages 85–92, 2005.

[110] Evren Sirin, Bijan Parsia, Dan Wu, James Hendler, and Dana Nau. HTN planning for Web Service composition using SHOP2. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):377–396, 2004.

[111] João Luís Sobrinho. Algebra and algorithms for QoS path computation and hop-by-hop routing in the Internet. In *20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, volume 10, pages 541–550, 2002.

[112] Shirin Sohrabi, Nataliya Prokoshyna, and Sheila A. McIlraith. Web service composition via generic procedures and customizing user preferences. In *5th International Semantic Web Conference (ISWC 2006)*, pages 597–611, 2006.

[113] Biplav Srivastava and Jana Koehler. Web Service Composition - Current Solutions and Open Problems. In *ICAPS 2003 workshop on Planning for Web Services*, pages 28–35, 2003.

[114] Anja Strunk. QoS-Aware Service Composition: A Survey. *IEEE European Conference on Web Services*, pages 67–74, 2010.

[115] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[116] Min Tian, Andreas Gramm, Tomasz Naumowicz, Hartmut Ritter, and Jochen H. Schiller. A concept for qos integration in web services. In *4th International Conference on Web Information Systems Engineering Workshops (WISE 2003)*, pages 149–155, 2003.

[117] Yves Vanrompay, Peter Rigole, and Yolande Berbers. Genetic algorithm-based optimization of service composition and deployment. In *Proceedings of the 3rd International Workshop on Services Integration in Pervasive Environments (SIPE'08)*, pages 13–18. ACM, 2008.

[118] W3C. Web Services Architecture. `http://www.w3.org/TR/ws-arch`, February 2004. Online; accessed 10 October 2015.

[119] Hiroshi Wada, Junichi Suzuki, Yuji Yamano, and Katsuya Oba. E^3: A Multiobjective Optimization Framework for SLA-Aware Service Composition. *IEEE Transactions on Services Computing*, 5(3):358–372, 2012.

[120] Florian Wagner, Fuyuki Ishikawa, and Shinichi Honiden. QoS-aware Automatic Service Composition by Applying Functional Clustering. In *IEEE International Conference on Web Services (ICWS)*, pages 89–96, 2011.

[121] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall, 2005.

[122] Thomas Weise. Web Service Challenge 2010. `http://www.it-weise.de/documents/files/W2010WSC_pres.pdf`, 2010. [Online; accessed May-2015].

[123] Thomas Weise, Steffen Bleul, Diana Comes, and Kurt Geihs. Different Approaches to Semantic Web Service Composition. In *2008 Third International Conference on Internet and Web Applications and Services*, pages 90–96, 2008.

[124] Thomas Weise, Steffen Bleul, Marc Kirchhoff, and Kurt Geihs. Semantic Web Service Composition for Service-Oriented Architectures. In *10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services*, pages 355–358, 2008.

[125] Bin Wu, Ying Li, Jian Wu, and Jianwei Yin. AWSP: An Automatic Web Service Planner based on Heuristic State Space Search. In *IEEE International Conference on Web Services (ICWS)*, pages 403–410, 2011.

[126] Zixin Wu, Karthik Gomadam, Ajith Ranabahu, Amit P. Sheth, and John A. Miller. Automatic Composition of Semantic Web Services using Process and Data Mediation. In *Proceedings of the 9-th International Conference on Enterprise Information Systems (ICEIS'07)*, pages 453–461, Funchal, Portugal, 2007.

[127] Zhou Xiangbing. Semantics Web Service Characteristic Composition Approach Based on Particle Swarm Optimization. volume 56 of *Lecture Notes in Electrical Engineering*, pages 279–287. Springer-Verlag, 2010.

[128] Jiuyun Xu, Kun Chen, and Stephan Reiff-Marganiec. Using Markov Decision Process Model with Logic Scoring of Preference Model to Optimize HTN Web Services Composition. *International Journal of Web Services Research*, 8(2):53–73, 2011.

[129] Yixin Yan, Bin Xu, and Zhifeng Gu. Automatic Service Composition Using AND/OR Graph. In *10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services*, pages 335–338, 2008.

[130] Yixin Yan, Bin Xu, Zhifeng Gu, and Sen Luo. A QoS-Driven Approach for Semantic Service Composition. In *IEEE International Conference on Commerce and Enterprise Computing*, pages 523–526, 2009.

[131] Tao Yu and Kwei-Jay Lin. Service selection algorithms for Web services with end-to-end QoS constraints. *Information Systems and e-Business Management*, 3(2):103–126, 2005.

[132] Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Transactions on the Web*, 1(1), 2007.

[133] Liangzhao Zeng, Boualem Benatallah, Anne HH Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. Qos-Aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.

[134] Xianrong Zheng and Yuhong Yan. An Efficient Syntactic Web Service Composition Algorithm Based on the Planning Graph Model. In *Proceedings of the 2008 IEEE International Conference on Web Services (ICWS 2008)*, pages 691–699, Washington, DC, USA, 2008. IEEE Computer Society.

[135] Zibin Zheng, Hao Ma, Michael R. Lyu, and Irwin King. Collaborative Web Service QoS Prediction via Neighborhood Integrated Matrix Factorization. *IEEE Transactions on Services Computing*, 6(3):289–299, 2013.

[136] Guobing Zou, Qiang Lu, Yixin Chen, Ruoyun Huang, You Xu, and Yang Xiang. QoS-Aware Dynamic Composition of Web Services Using Numerical Temporal Planning. *IEEE Transactions on Services Computing*, 7(1):18–31, 2014.

List of Algorithms

List of Figures

List of Tables