INTERNATIONAL DOCTORAL SCHOOL OF THE USC

Jacobo
Casas Ramos

PhD Thesis

Efficient alignment-based conformance checking of procedural and declarative process models

Santiago de Compostela, 2025

Doctoral Programme in Information Technology Research

INTERNATIONAL DOCTORAL SCHOOL OF THE USC

DOCTORAL THESIS

EFFICIENT ALIGNMENT-BASED CONFORMANCE CHECKING OF PROCEDURAL AND DECLARATIVE PROCESS MODELS

Author

Jacobo Casas Ramos

Supervisors: Manuel Lama Penín, Manuel Mucientes Molina

Tutor: Manuel Lama Penín

PHD PROGRAMME IN INFORMATION TECHNOLOGY RESEARCH

SANTIAGO DE COMPOSTELA

To my family
Constant sources of motivation

The greatest glory in living lies not in never falling, but in rising every time we fall.

Nelson Mandela

The important thing is not to stop questioning. Curiosity has its own reason for existing.

Albert Einstein

Acknowledgments

As I bring this chapter of my academic journey to a close, I am filled with an overwhelming sense of gratitude towards the countless individuals who have contributed to my growth, supported me through the highs and lows, and ultimately enabled me to complete this thesis. The pursuit of knowledge is often perceived as a solitary endeavor, but in reality, it is a testament to the power of collaboration, camaraderie, and community.

First and foremost, I would like to express my deepest gratitude to my thesis supervisors, Manuel Lama and Manuel Mucientes. Their guidance, expertise, and unwavering trust in me have been instrumental in shaping this research into what it is today. The freedom they afforded me to explore innovative ideas, coupled with their patient mentorship, has not only honed my academic skills but also instilled in me a sense of confidence and purpose. I am indebted to them for creating an environment that fostered growth, creativity, and intellectual curiosity.

I would also like to extend my appreciation to the Department of Electronics and Computing, as well as the Center for Research in Intelligent Technologies (CiTIUS) at the University of Santiago de Compostela. The academic environment, resources, and support provided by these institutions have been essential to the successful completion of this thesis. The tireless efforts of the CiTIUS support staff, often working behind the scenes, have greatly facilitated my research endeavors, and I am grateful for their dedication.

My research stay in Bolzano, under the guidance of Marco Montali, Alessandro Gianola, and Sarah Winkler, was a pivotal experience that broadened my perspectives, challenged my assumptions, and enriched my understanding of the field. The warmth, hospitality, and intellectual stimulation I received during my time at the KRDB Research Centre for Knowledge and Data have left an indelible mark on my academic journey.

This research would not have been possible without the financial support of various organizations. I am grateful to the Spanish Ministerio de Ciencia e Innovación for the grants

(PID2023-149549NB-I00, TED2021-130374B-C21, PID2020-112623GB-I00) that helped fund this project and were co-funded by the European Regional Development Fund (ERDF). Additionally, I acknowledge the support of the Galician Consellería de Cultura, Educación e Universidade (grant numbers ED431C 2018/29 and ED431G 2019/04) and the Spanish Ministerio de Universidades under the FPU national plan (grant number FPU19/06668). The funding provided by CiTIUS for my research stay has also been invaluable.

Lastly, I would like to express my heartfelt gratitude to my family, who have been my rock, my inspiration, and my safe haven throughout this journey. Their unwavering support, love, and encouragement have enabled me to pursue my passions with unrelenting dedication. Without their sacrifices, patience, and understanding, I would not be where I am today.

As I reflect on the people and experiences that have shaped this thesis, I am reminded of the power of human connection, collaboration, and community. This research is not just a testament to my own efforts but also a tribute to the collective contributions of those who have supported me along the way. I hope that this work will, in some small measure, contribute to the greater good and inspire others to pursue their passions with purpose, dedication, and perseverance.

June 6, 2025

Contents

Resumo

No complexo panorama das organizacións modernas, os procesos desempeñan un papel vital na consecución de obxectivos empresariais específicos [1]. Estes procesos poden atoparse en diversos campos como a xestión de servizos de TI [2], a optimización do rendemento deportivo [3, 4], a sanidade [5, 6, 7, 8] e o desenvolvemento de software [9, 10, 11]. Comprender como se executan os procesos na realidade, en lugar do seu deseño teórico, pode axudar significativamente a identificar oportunidades de mellora, como a redución de custos, a mellora da calidade e a mitigación do risco [1]. Por exemplo, nas industrias altamente reguladas, demostrar a conformidade dos procesos a través de rexistros de eventos detallados é esencial para cumprir os requisitos legais e regulatorios.

A importancia dos procesos esténdese máis alá da xestión operativa para abranguer dimensións estratéxicas, tácticas e de conformidade. A Xestión de Procesos de Negocio (*Business Process Management*, BPM) comprende un conxunto de metodoloxías, técnicas e ferramentas dirixidas ao deseño, execución e mellora continua dos procesos de negocio [12]. O ciclo de vida da BPM caracterízase por un marco iterativo que facilita o perfeccionamento constante do proceso [13]. Esta estrutura cíclica implica identificar o proceso, descubrir un modelo de proceso inicial, analizar os seus puntos débiles, redeseñar e implementar o proceso que resolve eses problemas e monitorizar os indicadores de rendemento relevantes para completar o ciclo.

A Minaría de Procesos (*Process Mining*) emerxeu como un novo campo de investigación que aproveita coñecementos baseados en datos para automatizar e optimizar a BPM [14]. Este enfoque aproveita a riqueza de información capturada por sistemas de software que apoian os procesos organizacionais, rexistrada como rexistros de eventos que capturan a pegada dixital das operacións dunha organización. Analizando os rexistros de eventos, as técnicas de Minaría de Procesos poden descubrir modelos de proceso que reflicten con precisión o

estado actual do proceso, comprobar a conformidade con novos rexistros de eventos para identificar discrepancias e desviacións e, finalmente, mellorar os modelos de proceso usando o coñecemento recollido das análises anteriores [15]. A clave para desbloquear este potencial reside no desenvolvemento de técnicas capaces de extraer coñecemento valioso dos rexistros de eventos.

A comprobación da conformidade (*conformance checking*) é unha área vital dentro da Minaría de Procesos, que se centra en comparar o comportamento observado nos rexistros de eventos cun modelo de proceso, co fin de identificar e avaliar desviacións e incongruencias. Entre as técnicas máis empregadas e que ofrecen unha información máis útil para esta tarefa atópase o cálculo de aliñamentos óptimos, xa que permite relacionar o comportamento real recollido nos rexistros de eventos co comportamento esperado definido polos modelos de proceso, mesmo cando non existe unha coincidencia perfecta [1]. Esta investigación de doutoramento centra a súa atención na comprobación da conformidade, abordando especialmente a complexidade e variabilidade inherentes aos procesos de negocio reais. As técnicas existentes adoitan ter dificultades para facer fronte á complexidade dos procesos do mundo real. Para superar esta limitación, o obxectivo é desenvolver métodos máis escalables, flexibles e eficientes para a comprobación da conformidade abordando as limitacións actuais e ofrecendo solucións prácticas que apoien as organizacións na mellora da comprensión, xestión e optimización dos seus procesos.

Os modelos de proceso son ferramentas esenciais para entender e analizar os procesos de negocio. Os modelos destilan a complexidade dun proceso nunha abstracción simplificada pero significativa, capturando as actividades clave e relacións [16]. En esencia, os modelos de proceso definen un conxunto de trazas válidas —secuencias finitas de eventos— dentro do proceso. Existen varios formalismos utilizados para representar os modelos de proceso, incluíndo Notación e Modelado de Procesos de Negocio (*Business Process Model Notation*, BPMN) [17], redes de Petri (*Petri nets*) e redes de fluxo de traballo [18], DECLARE [19], Gráficos de Seguimento Directo (*Directly-Follows Graphs*, DFGs) [20], Árbores de Proceso (*Process Trees*) [21], Redes Causais (*Causal nets*) [22] ou Gráficos Dinámicos de Condición Resposta (*Dynamic Condition Response Graphs*, DCR Graphs) [23].

A distinción entre modelos de proceso procedimentais e declarativos é un aspecto clave da Minaría de Procesos, pois inflúe fundamentalmente no enfoque adoptado para modelar e analizar os procesos de negocio. Os modelos de proceso procedimentais, como as redes de Petri, caracterízanse por definir explicitamente o fluxo de control, que dicta a secuencia de actividades

e decisións dentro dun proceso [24]. Este enfoque proporciona unha descrición detallada e paso a paso de como se debe executar un proceso. Isto facilita modelar e analizar procesos estritamente controlados, como os vinculados a acordos de nivel de servizo (*service-level agreements*). Con todo, os modelos procedimentais poden volverse excesivamente complexos e engorrosos, especialmente ao tratar con procesos grandes e intrincados. A creación destes modelos require unha experiencia e esforzo significativos, o que pode ser unha desvantaxe importante.

Por outra banda, os modelos de proceso declarativos como DECLARE ofrecen un enfoque máis flexible e adaptable ao modelado de procesos [25]. Ao centrarse na declaración de restricións e regras que gobernan o comportamento do proceso, en lugar de definir explicitamente o fluxo de control, os modelos declarativos proporcionan unha representación máis abstracta e de alto nivel dos procesos de negocio. Este enfoque permite unha maior flexibilidade e dinamismo, pois non impón unha estrutura ríxida ao proceso. Os modelos declarativos son particularmente adecuados para modelar e analizar procesos de negocio dinámicos e flexibles, onde os camiños de execución son altamente variables ou están suxeitos a cambios frecuentes.

A principal diferenza entre os modelos de procesos procedimentais e os modelos declarativos reside na filosofía de deseño do comportamento do proceso. Os modelos procedimentais prescriben exactamente *como* debe executarse un proceso, mentres que os modelos declarativos especifican *que* restricións deben cumprirse durante a súa execución [26]. Esta diferenza fundamental ten importantes implicacións para a análise e mellora dos procesos de negocio. Os modelos declarativos permiten un enfoque máis modular e incremental do modelado de procesos, onde se poden engadir ou eliminar facilmente novas restricións e regras segundo sexa necesario. En contraste, os modelos procedimentais poden ser máis axeitados para procesos ríxidos e estables, como os que se atopan nos ámbitos legal, regulador ou financeiro, nos que a adhesión a protocolos e procedementos establecidos resulta crucial. A elección entre modelos de procesos procedimentais e declarativos depende, en última instancia, dos requisitos e características específicos do proceso de negocio que se desexe modelar [16].

A integración de datos desempeña un papel valioso na xestión moderna de procesos de negocio, pois permite ás organizacións capturar as relacións complexas entre os procesos e os datos [27]. Os modelos de proceso conscientes de datos (*data-aware process models*) estenden os enfoques tradicionais de modelado de procesos incorporando atributos de datos e as súas relacións con actividades do proceso, permitindo unha comprensión máis completa dos procesos de negocio. Este enfoque recoñece que os procesos de negocio a miúdo están

impulsados por datos e que o fluxo de datos entre actividades pode ter un impacto significativo no comportamento do proceso. *Multi-Perspective* DECLARE (MP-DECLARE) é unha extensión da linguaxe DECLARE que permite incorporar condicións baseadas en datos nos modelos de proceso declarativos [28]. MP-DECLARE habilita aos modeladores a declarar restricións e regras baseadas en atributos de datos.

A comprobación da conformidade é un compoñente clave da Minaría de Procesos, que recibe modelos de proceso descubertos ou deseñados manualmente e un rexistro de eventos, e produce coñecementos clave necesarios para a mellora do proceso ou a toma de decisións baseadas en datos [29]. As técnicas de execución baseadas en fichas (*token-based replay*) [30, 31] a miúdo empréganse para a comprobación da conformidade, pero carecen da capacidade de detección da causa raíz, o que limita a calidade dos coñecementos que poden proporcionar. Como resultado, a pesar de ofrecer vantaxes en termos de velocidade, as técnicas de reprodución baseadas en fichas poden non ofrecer o mesmo nivel de información diagnóstica que os métodos baseados en aliñamentos. Os aliñamentos óptimos [32], por outra banda, proporcionan unha comprensión máis completa dos problemas de conformidade ao establecer unha conexión entre o comportamento real observado no rexistro de eventos e unha traxectoria factible a través do modelo de proceso. Estes enfoques baseados en aliñamentos emerxeron como as solucións máis exitosas para a comprobación da conformidade debido á súa capacidade de proporcionar diagnósticos precisos [29].

Os aliñamentos son extremadamente útiles porque identifican e listan explicitamente todas as discrepancias entre o comportamento real rexistrado nos rexistros de eventos e o comportamento esperado definido nos modelos de proceso. Un aliñamento é unha secuencia de movementos que concilia os eventos nunha traza cunha execución válida do modelo de proceso. Estes movementos comezan ao principio da traza e o estado inicial do modelo de proceso, progresando paso a paso a través de ambos ata alcanzar o final da traza e o modelo. Hai tres tipos de movimentos: (i) movementos sincrónicos, onde un evento na traza coincide cun paso esperado no modelo, indicando comportamento correcto; (ii) movementos de rexistro, que ocorren cando un evento está presente na traza pero non ten un paso correspondente no modelo; e (iii) movementos de modelo, que ocorren cando o modelo espera unha actividade que non está presente na traza. A cada movemento asígnase un custo, normalmente con movementos sincrónicos tendo un custo de cero, e movementos de rexistro ou modelo indicando desviacións e, por tanto, incurrindo nun custo de un. O obxectivo é atopar o aliñamento óptimo, é dicir, a secuencia de movementos co custo total mínimo que explica mellor como se axusta (ou desvía)

o comportamento observado ao comportamento modelado.

Desenvolver algoritmos eficientes para computar aliñamentos óptimos é crucial para afrontar os desafíos inherentes á comprobación da conformidade baseada en aliñamentos. O algoritmo A* é unha enfoque amplamente adoptado debido á súa optimalidade e completude [33]. Con todo, a súa aplicación efectiva neste contexto depende do deseño dunha función heurística apropiada. A heurística debe equilibrar a precisión das estimacións de custo e o tempo computacional incurrido por estado para garantir un bo rendemento xeral. Mellorar o estado da arte require o desenvolvemento de heurísticas máis sofisticadas que manteñan este equilibrio. Ademais, o rendemento pode mellorarse significativamente reducindo o número de estados explorados polo algoritmo A* a través de técnicas como poda de estados sen saída e a reformulación do problema —áreas que aínda están pouco exploradas nos traballos actuais.

Esta investigación ten como obxectivo explorar novas metodoloxías e técnicas para aliñar eficientemente rexistros de eventos con modelos de proceso procedimentais, declarativos e declarativos conscientes de datos. Preténdese incrementar substancialmente o rendemento e a escalabilidade dos algoritmos actuais de comprobación da conformidade, permitindo unha análise máis rápida e precisa de grandes e complexos rexistros de eventos atopados en ambientes reais [14]. En última instancia, isto contribuirá a unha maior eficiencia dos esforzos de Minaría de Procesos en múltiples dominios.

A hipótese de investigación que guía este doutoramento é que a exploración de diversas técnicas de optimización, incluíndo heurísticas avanzadas A*, novas estratexias de poda do espazo de busca e reformulacións innovadoras do problema, é posible mellorar significativamente a eficiencia do cálculo de aliñamentos óptimos, permitindo así o cálculo de aliñamentos complexos en procesos reais que actualmente resultan intratábeis cos métodos máis avanzados dispoñibles. Para investigar esta hipótese, defínese un conxunto de obxectivos de investigación que, de seren acadados, poderían demostrar a viabilidade de mellorar de forma significativa tanto a eficiencia como a escalabilidade do cálculo de aliñamentos óptimos. Estes obxectivos procuran abordar os desafíos de rendemento presentes nos distintos paradigmas de modelado de procesos: modelos procedimentais, modelos declarativos e modelos declarativos conscientes dos datos.

As principais contribucións desta tese inclúen un algoritmo óptimo de aliñamento eficiente para modelos de proceso procedimentais [34], unha nova abordaxe para a comprobación da conformidade de modelos de proceso declarativos [35] e técnicas avanzadas de comprobación da conformidade para modelos de proceso conscientes de datos con restricións de datos

flexibles [36]. Estas contribucións detállanse en Chapters 2 to 4 respectivamente e resúmense a continuación.

REACH: Researching Efficient Alignment-based Conformance Checking

No contexto da comprobación da conformidade, os modelos de proceso procedimentais, particularmente as redes de Petri, gañaron unha atención significativa debido á súa capacidade para representar con precisión os procesos de negocio [24]. As redes de Petri son un formalismo amplamente utilizado para modelar procesos de negocio, e a súa aplicación na comprobación da conformidade foi extensivamente estudada [32, 37, 38, 39, 40, 41, 42]. Porén, o cálculo de aliñamentos óptimos entre rexistros de eventos e modelos de rede de Petri segue sendo unha tarefa desafiante, especialmente ao tratar con modelos grandes e complexos. A dificultade reside na complexidade computacional requirida para atopar estes aliñamentos, o que pode resultar en tempos de procesamento prolongados e incluso impedir o cálculo dalgúns aliñamentos. O algoritmo A* emerxeu como unha opción destacada para esta tarefa debido á súa optimalidade e completude [33].

Para abordar este reto, propónse un algoritmo eficiente chamado REACH , que aproveita o algoritmo A* e técnicas de optimización para mellorar o rendemento e a escalabilidade no cálculo de aliñamentos óptimos entre rexistros de eventos e redes de Petri. A aplicación do algoritmo A* no cálculo de aliñamentos óptimos require unha consideración coidadosa da función heurística utilizada para guiar a busca. Unha heurística ben deseñada pode mellorar significativamente o rendemento do algoritmo reducindo o número de estados que deben ser explorados.

Neste contexto, propónse unha nova heurística que atopa rapidamente as actividades requiridas e as compara coa parte restante da traza. As actividades requiridas son aquelas que deben executarse para alcanzar o final do modelo, e a súa identificación pode proporcionar coñecementos valiosos sobre a conformidade do proceso. Ao comparar estas actividades coa traza restante, a heurística pode proporcionar estimacións de custo máis precisas e acelerar o algoritmo sen perder a súa optimalidade.

Ademais, as técnicas para reducir o número de estados explorados polo algoritmo A* son esenciais para mellorar o rendemento. O algoritmo proposto introduce técnicas para eliminar do espazo de busca os estados innecesarios. Especificamente, empréganse dúas optimizacións:

Redución de Estados forzando movementos asincrónicos do Modelo (*SRModel*) e Redución de Estados forzando movementos asincrónicos do Rexistro (*SRLog*). *SRModel* comproba se as transicións habilitadas do modelo non poden executarse na traza restante e forza un movemento de modelo se é necesario. *SRLog* identifica actividades vivas no modelo que poden executarse en calquera punto no futuro e forza un movemento asincrónico do rexistro cando a seguinte actividade na traza non coincide con ningunha destas actividades vivas. Ademais, o algoritmo proposto utiliza un algoritmo voraz que devolve un aliñamento subóptimo como un límite superior de custo para o algoritmo principal, filtrando estados que non levarán ao aliñamento óptimo.

Ademais, o algoritmo utiliza un gráfico de acadabilidade parcial (*Partial Reachability Graph*, PRG) para acelerar a execución dos modelos de proceso durante o cálculo do aliñamento. O PRG constrúese dinamicamente á medida que se alcanzan novos estados de execución do modelo, permitindo un acceso máis rápido a transicións habilitadas e novos estados de execución. Esta optimización reduce a complexidade computacional asociada coa execución dos modelos de proceso, especialmente cando se aliñan múltiples trazas ao modelo, xa que o PRG se reutiliza entre diferentes execucións do algoritmo, logrando melloras significativas no rendemento.

A proposta foi implementada completamente e sometida a probas exhaustivas. Realizáronse experimentos exhaustivos para avaliar o rendemento de REACH fronte a 10 algoritmos de comprobación da conformidade de vangarda utilizando un conxunto de datos de 227 pares rexistro-modelo de diversos dominios, incluíndo sanidade, finanzas e xestión de servizos de TI. Os resultados demostran que REACH supera aos algoritmos de comprobación da conformidade de vangarda tanto en tempo de cálculo como no número de problemas resoltos. En particular, REACH resolve o 95% dos pares rexistro-modelo probados en menos de 45 segundos, mentres que outros algoritmos do estado da arte non conseguen aliñar tantos pares nin sequera cun límite de tempo de 300 segundos.

En resumo, as melloras de rendemento logradas por este traballo permiten o cálculo eficiente de aliñamentos óptimos para modelos de proceso de rede de Petri. Ao eliminar a necesidade de confiar en métodos rápidos pero menos precisos, o algoritmo proposto abre novas posibilidades para a aplicación de técnicas de comprobación da conformidade avanzadas en escenarios do mundo real.

Jacobo Casas Ramos

DeclareAligner: A Leap Towards Efficient Optimal Alignments for Declarative Process Model Conformance Checking

Os modelos de proceso declarativos ofrecen un enfoque máis flexible para modelar procesos de negocio, facéndoos máis axeitados para ambientes dinámicos. A diferenza dos modelos procedimentais, os modelos declarativos non definen explicitamente o fluxo de control, senón que especifican *qué* restricións deben cumprirse durante a execución do proceso sen prescribir exactamente *como* debe facerse [26]. A flexibilidade dos modelos de proceso declarativos introduce novos desafíos para a comprobación da conformidade, xa que o espazo de busca de posibles aliñamentos aumenta significativamente.

DECLARE é a linguaxe principal para o modelado de procesos declarativos, ofrecendo unha forma flexible e concisa de definir restricións sobre procesos de negocio mediante a utilización de plantillas de restricións que se completan con actividades relevantes [43]. Por exemplo, considere o proceso de alta hospitalaria dun paciente, que inclúe actividades como a prescrición de medicamentos, a alta do paciente e a programación da seguinte cita. Un modelo DECLARE pode especificar restricións para este proceso, como "un paciente debe recibir medicamentos antes de ser dado de alta" ou "debe programarse unha seguinte cita despois da alta, sen que haxa outra alta no medio". Isto permite certa flexibilidade no proceso, ao tempo que se impoñen restricións fundamentais.

Cando se computan aliñamentos óptimos para modelos de proceso declarativos, os algoritmos do estado da arte adoitan ter problemas de escalabilidade e eficiencia. O estado da arte [44, 45, 46] traduce a linguaxe DECLARE en fórmulas de Lóxica Temporal Lineal sobre trazas finitas (LTLf) e posteriormente a autómatas de estados finitos [47]. Estes autómatas poden ser utilizados para computar aliñamentos utilizando algoritmos semellantes aos utilizados para modelos procedimentais. Con todo, este proceso de tradución pode levar a unha perda de simplicidade semántica e a un aumento da complexidade computacional. Especificamente, o proceso de tradución pode ocultar a natureza declarativa das restricións orixinais, polo que se perden potenciais optimizacións que poderían ser aproveitadas se a representación declarativa se utilizase directamente para obter aliñamentos óptimos.

Para abordar este reto, proponse DECLAREALIGNER, un algoritmo novidoso que estende o algoritmo de busca A* para abordar o problema desde unha perspectiva fresca, aproveitando a flexibilidade dos modelos declarativos utilizándoos directamente durante o proceso de aliñamento. O algoritmo constrúe o espazo de busca de forma incremental a medida que

se explora, comezando cun estado inicial que respresenta toda a traza. A partir desta base, xéranse novos estados iterativamente mediante a aplicación de accións correctivas específicas, producindo finalmente un estado obxectivo que corresponde a unha execución válida do proceso. Ao comparar a traza orixinal cunha traza válida representada por este estado obxectivo, pode extraerse un aliñamento óptimo, resaltando as desviacións entre o comportamento real e o esperado.

O algoritmo proposto ofrece varias innovacións significativas que melloran o seu rendemento e escalabilidade. Só considera tomar accións que contribúan directamente a reparar restricións violadas, e propón varias estratexias de optimización da busca, incluíndo unha heurística que combina as accións suxeridas para todas as restricións violadas dun estado para proporcionar estimacións de custo precisas. Ademais, elimínanse estados que non poden alcanzar o aliñamento óptimo para eliminar pólas infrutíferas do espazo de busca, e aplícanse antes da busca accións requiridas por restricións de tipo cadea que non interferen con outras restricións. Estas innovacións permiten a DeclareAligner obter aliñamentos óptimos para modelos de proceso declarativos de forma eficiente, converténdoo nunha ferramenta valiosa para a comprobación da conformidade en contornas dinámicas de negocio. O soporte nativo do algoritmo para modelos declarativos aproveitando a súa flexibilidade e as súas estratexias de optimización fan que sexa axeitado para manexar modelos de proceso complexos e rexistros de eventos grandes.

A avaliación de DeclareAligner realízase sobre un conxunto de datos exhaustivo que contén 8.054 pares traza-modelo, incluíndo datos reais e sintéticos. Preséntase un caso de estudo para demostrar a utilidade práctica do algoritmo proposto. Ademais, realízase un estudo de ablación para avaliar o impacto de cada compoñente de DeclareAligner no seu rendemento global. Os resultados mostran que a poda temperá, o preprocesamento de restricións e as reparacións agrupadas son compoñentes clave que contribúen significativamente á eficiencia do algoritmo. Finalmente, realízase unha comparación detallada fronte ao estado da arte. DeclareAligner alcanza un tempo de execución medio de $7{,}6$ segundos para o conxunto de datos completo, o que é substancialmente menor que o seguinte mellor algoritmo (DeclareReplayer) cun tempo de execución medio de $72{,}5$ segundos. Ademais, as optimizacións propostas permiten a DeclareAligner obter aliñamentos con éxito para 231 pares traza-modelo que permanecen sen resolver por todos os demais algoritmos dentro dun límite de tempo de 5 minutos.

En resumo, DeclareAligner ofrece unha forma innovadora de cómputo de aliñamentos

óptimos para modelos de proceso declarativos, aproveitando a flexibilidade destes modelos e introducindo varias estratexias de optimización para mellorar o rendemento e a escalabilidade. A capacidade do algoritmo para computar aliñamentos óptimos de forma eficiente convérteo nunha ferramenta valiosa para a comprobación da conformidade en contornas empresariais dinámicas, e a súa avaliación demostra a súa superioridade respecto do estado da arte.

Efficient Conformance Checking of Rich Data-Aware Declare Specifications

As seccións anteriores centráronse no aspecto do fluxo de control, proporcionando unha base de alto rendemento para a comprobación da conformidade dentro da Minaría de Procesos. Con todo, á medida que os procesos vólvense cada vez máis complexos e impulsados por datos, hai unha crecente necesidade de estender a comprobación da conformidade a modelos declarativos conscientes de datos. O modelado declarativo destaca pola súa flexibilidade, que desafortunadamente amplía o espazo de busca de aliñamentos e complica a comprobación da conformidade. A introdución de datos a nivel de evento aumenta aínda máis esta complexidade, xa que se require a consideración simultánea das perspectivas do fluxo de control e dos datos. A pesar destes retos, os modelos conscientes de datos proporcionan coñecementos máis ricos e sensibles ao contexto, xustificando así a necesidade do aumento do esforzo computacional.

A necesidade de restricións multiperspectiva máis expresivas no modelado de procesos está ben establecida [48, 49, 50]. Considere, por exemplo, un escenario de envío gobernado por unha restrición de resposta DECLARE entre os eventos "Envío de Paquete" (A) e "Confirmación de Entrega" (B): sempre que ocorre un envío, debe recibirse unha confirmación de entrega correspondente. Con todo, o momento desta confirmación depende de condicións adicionais. Se o paquete pesa menos de $10{,}5$ kg e está destinado a unha rexión xeográfica específica, a confirmación debe recibirse dentro dos 3 días seguintes ao envío. De non ser así, o prazo amplíase a 10 días. Un algoritmo que considere só a perspectiva do fluxo de control non podería impor estas regras temporais e dependentes dos datos, pasando por alto así aspectos críticos das garantías de nivel de servizo incorporadas no proceso empresarial.

MP-DECLARE é a linguaxe líder de modelado de procesos declarativos, ofrecendo unha forma flexible e concisa de definir restricións sobre procesos empresariais que integran tanto o fluxo de control como os aspectos de datos [51]. Con todo, o deseño dunha comprobación de conformidade baseada en aliñamentos para especificacións declarativas enriquecidas con

datos segue sendo en gran parte un problema aberto. Os traballos anteriores centráronse só no fluxo de control, utilizando representacións como gráficos de Resposta de Condición Dinámica (*Dynamic Condition-Response Graphs*, DCR Graphs) [52], restricións DECLARE [53] ou Lóxica Temporal Lineal sobre trazas finitas (LTLf) [54]. Ata onde sabemos, o único traballo existente sobre aliñamento consciente de datos que utiliza DECLARE é [28]. Con todo, este traballo está limitado a datos numéricos e comparacións básicas entre variables e constantes —por exemplo, x > 5—, sen soporte para comparacións entre variables de eventos diferentes, unha limitación importante que o noso traballo pretende superar.

Para afrontar este reto, propónse Data-Aware DECLARE Aligner, un enfoque innovador que integra dúas estratexias complementarias: un mecanismo de busca A* para explorar o espazo de posibles aliñamentos e a resolución por Satisfiability Modulo Theories (SMT) para xestionar restricións de fluxo de control e datos [55]. A SMT é un poderoso problema de decisión que incorpora varias teorías de fondo, como a aritmética e as funcións non interpretadas, para determinar a satisfacibilidade de fórmulas lóxicas. Isto converte a SMT nunha ferramenta prometedora para o cálculo de aliñamentos conscientes de datos, especialmente en dominios complexos onde se involucran tipos de datos ricos e relacións intricadas entre variables.

Data-Aware DECLARE Aligner define un espazo de busca cuxo estado inicial codifica a traza orixinal como unha fórmula SMT. Os veciños xéranse aplicando accións sobre un estado que contribúen directamente a reparar unha restrición violada do modelo de proceso. Cada estado explorado sométese a unha detección de violacións baseada en SMT, e calquera discrepancia desencadea a xeración de estados fillos a través de reparacións específicas. O algoritmo avanza ata alcanzar un estado obxectivo cun custo de reparación mínimo e sen violacións, a partir do cal se pode reconstruír o aliñamento óptimo. Demóstrase a corrección do método proposto, e experimentos exhaustivos amosan a súa eficiencia, incluso con condicións de datos complexas.

O algoritmo proposto trata eficazmente as restricións complexas sobre datos aproveitando a potencia da resolución mediante SMT. Trátanse as marcas de tempo como calquera outro atributo de datos, o que permite un razoamento simultáneo sobre restricións de fluxo de control e de datos. Este tratamento unificado tamén facilita a especificación e aplicación de restricións de correlación temporal. Data-Aware DECLARE Aligner optimizouse para mellorar o rendemento, empregándose varias estratexias clave para reducir a complexidade computacional. Unha optimización importante consiste en seleccionar unha violación para reparar, onde o algoritmo elixe a violación que resulta no menor número de estados fillos, retrasando así a explosión do número de estados e mellorando a eficiencia da busca. Ademais, detéctanse os estados

sen saída precalculando todas as violacións e comprobando se hai reparos aplicables para todas; se non é así, o estado é eliminado do espazo de busca, eliminando así pólas infrutíferas. Tamén se identifican e eliminan estados insatisfactibles, nos que se introduciron condicións incompatibles durante a xeración do estado.

A avaliación do algoritmo proposto demostra a súa eficiencia. Realizáronse experimentos sobre un conxunto de datos con diferentes niveis de complexidade. Inicialmente, usáronse condicións de datos sinxelas para a activación e obxectivo das restricións, a fin de asegurar a compatibilidade co único algoritmo comparable [28]. En media, Data-Aware DECLARE Aligner aliña trazas 2,9 veces máis rápido que o método do estado da arte. Ademais, a avaliación resalta a capacidade do algoritmo para xestionar dependencias de datos intricadas, como condicións de correlación, estendendo o conxunto de datos orixinal con restricións significativamente máis complexas.

En conclusión, o algoritmo proposto ofrece unha solución novedosa ao reto da comprobación eficiente da conformidade para modelos de proceso declarativos conscientes de datos. Ao integrar a busca A* e a resolución SMT, o algoritmo define un espazo de busca que soporta tipos de datos ricos e expresións de datos complexas, facendo posible xestionar condicións de datos intricadas non soportadas polo estado da arte.

Conclusións

Este doutoramento fixo contribucións significativas ao campo da Minaría de Procesos mediante a investigación da computación eficiente de aliñamentos óptimos entre rexistros de eventos e modelos de proceso. A motivación principal detrás desta investigación provén do recoñecemento de que as técnicas de comprobación da conformidade existentes a miúdo teñen dificultades para computar aliñamentos óptimos de forma eficiente para procesos reais, particularmente en escenarios nos que as trazas dos rexistros de eventos son grandes ou os modelos de proceso son complexos.

Os principais descubrimentos desta investigación demostran a eficiencia dos algoritmos propostos no cálculo de aliñamentos óptimos entre rexistros de eventos e modelos de proceso procedimentais, declarativos e declarativos conscientes de datos. Os resultados teñen implicacións significativas para o campo da Minaría de Procesos, permitindo así que as organizacións obteñan coñecementos máis profundos sobre os seus procesos, identifiquen áreas de mellora e optimicen as súas operacións con maior confianza.

Este traballo presentou un marco integral para a comprobación da conformidade baseada en aliñamentos eficiente, combinando fundamentos formais con implementacións completas e avaliación exhaustiva sobre conxuntos de datos reais e sintéticos. Os algoritmos propostos demostraron superar os métodos existentes do estado da arte, mostrando melloras significativas en eficiencia e flexibilidade. En última instancia, este doutoramento acadou todos os seus obxectivos con éxito, abrindo o camiño para unha nova xeración de técnicas de Minaría de Procesos e consolidando un impacto duradeiro no campo.

Summary

In the complex landscape of modern organizations, processes play a vital role in achieving specific business objectives [1]. These processes can be found in diverse fields such as IT service management [2], athletic performance optimization [3, 4], healthcare [5, 6, 7, 8], and software development [9, 10, 11]. Understanding how processes are executed in reality, as opposed to their theoretical design, can significantly aid in identifying opportunities for improvement, such as cost reduction, quality enhancement, and risk mitigation [1]. For instance, in highly regulated industries, demonstrating process compliance through detailed event logs is essential for meeting legal and regulatory requirements.

The significance of processes extends beyond operational management to encompass strategic, tactical, and compliance dimensions. Business Process Management (BPM) encompasses a range of methodologies, techniques, and tools aimed at designing, executing, and continuously improving business processes [12]. The BPM lifecycle is characterized by an iterative framework that facilitates ongoing process improvement [13]. This cyclical structure involves identifying the process, discovering an initial process model, analyzing it for weaknesses, redesigning and implementing the process that fixes those issues, and monitoring relevant performance metrics to complete the cycle.

Process Mining has emerged as a new field of research that leverages data-driven insights to automate and optimize BPM [14]. This approach harnesses the wealth of information captured by software systems that support organizational processes, recorded as event logs that capture the digital footprint of an organization's operations. By analyzing event logs, Process Mining techniques can discover process models that accurately reflect the current state of the process, check conformance with new event logs to identify discrepancies and deviations, and ultimately enhance process models using knowledge gathered from previous analyses [15]. The key to unlocking this potential lies in developing techniques that can extract valuable knowledge from

event logs.

One of the main challenges in Process Mining is conformance checking, which focuses on comparing the behavior observed in event logs with a process model to identify and assess deviations and inconsistencies. Among the most widely used and insightful techniques for conformance checking is the computation of optimal alignments, as they relate the actual behavior captured in event logs to the expected behavior defined by process models, even in the absence of a perfect match [1]. This PhD research focuses on advancing conformance checking, particularly addressing the complexity and variability inherent in real-life business processes. Existing techniques often struggle to cope with the complexity of real-world processes. To bridge this gap, the goal is to develop more scalable, flexible, and efficient methods for conformance checking by tackling current limitations and offering practical solutions that support organizations in better understanding, managing, and optimizing their processes.

Process models are essential tools for understanding and analyzing business processes. They distill the complexity of a process into a simplified yet meaningful abstraction, capturing key activities and relationships [16]. In essence, process models define a set of valid traces —finite sequences of events— within the process. There are various formalisms used to represent process models, including Business Process Model Notation (BPMN) [17], Petri nets and workflow nets [18], DECLARE [19], Directly-Follows Graphs (DFGs) [20], Process Trees [21], Causal nets [22], or Dynamic Condition Response Graphs (DCR Graphs) [23].

The distinction between procedural and declarative process models is a key aspect of Process Mining, as it fundamentally influences the approach to modeling and analyzing business processes. Procedural process models, such as Petri nets, are characterized by their explicit definition of control flow, which dictates the sequence of activities and decisions within a process [24]. This approach provides a detailed, step-by-step description of how a process should be executed. It makes it easier to model and analyze tightly controlled processes, such as those associated with service-level agreements. However, procedural models can become overly complex and cumbersome, particularly when dealing with large and intricate processes. The creation of these models demands significant expertise and effort, which can be a major drawback.

On the other hand, declarative process models like DECLARE offer a more flexible and adaptable approach to process modeling [25]. By focusing on the declaration of constraints and rules that govern process behavior, rather than explicitly defining control flow, declarative models provide a more abstract and high-level representation of business processes. This

approach allows for greater flexibility and dynamicity, as it does not impose a rigid structure on the process. Declarative models are particularly well-suited for modeling and analyzing dynamic and flexible business processes, where the execution paths are highly variable or subject to frequent changes.

The key difference between procedural and declarative process models lies in their perspective on process behavior. Procedural models prescribe exactly *how* a process should be executed, whereas declarative models specify *what* constraints must be satisfied during process execution [26]. This fundamental difference has significant implications for the analysis and improvement of business processes. Declarative models enable a more modular and incremental approach to process modeling, where new constraints and rules can be easily added or removed as needed. In contrast, procedural models may be better suited for strict and stable processes, such as those found in legal, regulatory, or financial domains, where adherence to established protocols and procedures is crucial. The choice between procedural and declarative process models ultimately depends on the specific requirements and characteristics of the business process being modeled [16].

Data integration plays a valuable role in modern business process management, as it enables organizations to capture the complex relationships between processes and data [27]. Data-aware process models extend traditional process modeling approaches by incorporating data elements and their relationships with process activities, allowing for a more comprehensive understanding of business processes. This approach recognizes that business processes are often driven by data, and that the flow of data between activities can have a significant impact on process behavior. Multi-Perspective DECLARE (MP-DECLARE) is an extension of the DECLARE language that allows for the incorporation of data conditions into declarative process models [28]. MP-DECLARE enables modelers to declare constraints and rules that are based on data attributes.

Conformance checking is a key component of Process Mining, which receives discovered or manually-designed process models and an event log, and outputs key insights required for process enhancement or data-driven decision-making [29]. Token-based replay techniques [30, 31] are often employed for conformance checking, but they lack the capability for root cause detection, which limits the quality of insights they can provide. As a result, despite their potential speed advantages, token-based replay techniques may not offer the same level of diagnostic information as alignment-based methods. Optimal alignments [32], on the other hand, provide a more comprehensive understanding of conformance issues by establishing a connection between the actual behavior observed in the event log and a feasible path through

the process model. Alignment-based approaches have emerged as the most successful solutions for conformance checking due to their ability to provide accurate diagnostics [29].

Alignments are extremely useful as they identify and explicitly list all discrepancies between the actual behavior recorded in event logs and the expected behavior defined in process models. An alignment is a sequence of moves that reconciles events in a trace with a valid execution of the process model. These moves begin at the start of the trace and the initial state of the process model, progressing step-by-step through both until the end of the trace and the model is reached. There are three types of moves: (i) synchronous moves, where an event in the trace matches an expected step in the model, indicating correct behavior; (ii) log moves, which occur when an event is present in the trace but has no corresponding step in the model; and (iii) model moves, which occur when the model expects an activity that is not present in the trace. Each move is assigned a cost, typically with synchronous moves having a cost of zero, and log or model moves indicating deviations and therefore incurring a cost of one. The objective is to find the optimal alignment, i.e., the sequence of moves with the minimum total cost that best explains how the observed behavior fits (or deviates from) the modeled behavior.

Developing efficient algorithms for computing optimal alignments is crucial to address the challenges inherent in alignment-based conformance checking. A* search is a widely adopted approach due to its optimality and completeness [33]. However, its effective application in this context hinges on the design of an appropriate heuristic function. The heuristic must strike a balance between the accuracy of cost estimates and the computational overhead incurred per state to ensure overall performance. Enhancing the state-of-the-art requires the development of more sophisticated heuristics that maintain this balance. Additionally, performance can be significantly improved by reducing the number of states explored by the A* algorithm through techniques such as dead-end pruning and problem reformulation—areas that remain underutilized in current approaches.

This research aims to explore novel methodologies and techniques for efficiently aligning event logs with procedural, declarative, and data-aware declarative process models. The goal is to significantly improve the performance and scalability of the current conformance checking algorithms, enabling faster and more accurate analysis of large and complex event logs found in real-world settings [14]. Ultimately, this will enhance the overall efficiency of Process Mining efforts in various domains.

The research hypothesis guiding this PhD is that exploring various optimization techniques, including advanced A* heuristics, novel search space pruning strategies, and innovative problem

reformulations, can significantly improve the efficiency of computing optimal alignments, thereby enabling the computation of complex alignments in real-life processes that are currently intractable with state-of-the-art methods. To investigate the stated hypothesis, a set of research objectives is defined that, if achieved, could demonstrate the feasibility of significantly improving the efficiency and scalability of computing optimal alignments. These objectives aim to address the performance challenges across different process modeling paradigms: procedural models, declarative models, and data-aware declarative models.

The main contributions of this thesis include an efficient optimal alignment algorithm for procedural process models [34], a novel approach for declarative process model conformance checking [35], and advanced conformance checking for data-aware declarative process models with flexible data constraints [36]. These contributions are detailed in Chapters 2 to 4 respectively and summarized next.

REACH: Researching Efficient Alignment-based Conformance Checking

In the context of conformance checking, procedural process models, particularly Petri nets, have gained significant attention due to their ability to accurately represent business processes [24]. Petri nets are a widely used formalism for modeling business processes, and their application in conformance checking has been extensively explored [32, 37, 38, 39, 40, 41, 42]. However, computing optimal alignments between event logs and Petri net models remains a challenging task, especially when dealing with large and complex models. The difficulty lies in the computational complexity required to find these alignments, which can result in extended processing times and even prevent the computation of some alignments. A* search has emerged as a prominent choice for this task due to its optimality and completeness [33].

To address this challenge, an efficient algorithm called REACH is proposed, which leverages A* search and optimization techniques to improve performance and scalability in computing optimal alignments between event logs and Petri net process models. The application of A* search in computing optimal alignments requires careful consideration of the heuristic function used to guide the search. A well-designed heuristic can significantly improve the performance of the algorithm by reducing the number of states that need to be explored.

In this context, a new heuristic that quickly finds required activities and compares them with the remaining trace is proposed. Required activities are those that must be executed to reach the

end of the model, and their identification can provide valuable insights into the conformance of the process. By comparing these activities with the remaining trace, the heuristic can provide more accurate cost estimates and speed up the algorithm without losing its optimality.

Furthermore, techniques for reducing the number of states explored by the A* algorithm are essential to improve performance. The proposed algorithm introduces techniques for pruning unnecessary states from the search space. Specifically, two optimizations are employed: States Reduction forcing asynchronous Model moves (SRModel) and States Reduction forcing asynchronous Log moves (SRLog). SRModel checks whether the enabled model transitions cannot be executed in the remaining trace and forces a model move if necessary. SRLog identifies alive activities in the model that can be executed at any point in the future and forces an asynchronous log move when the next activity in the trace does not match any of these alive activities. Additionally, a greedy algorithm that returns a sub-optimal alignment can be used as an upper bound of cost for the main algorithm, filtering states that will not lead to the optimal alignment.

Additionally, the algorithm utilizes a partial reachability graph (PRG) to speed up the execution of process models during alignment computation. The PRG is constructed dynamically as new model markings are reached, allowing for faster access to enabled transitions and new markings. This optimization reduces the computational complexity associated with executing process models, especially if multiple traces are aligned to the process models since the PRG is reused across different runs of the algorithm, resulting in significant performance improvements.

The proposal has been fully implemented and tested thoroughly. Extensive experiments have been conducted to evaluate the performance of REACH against 10 state-of-the-art conformance checking algorithms using a dataset of 227 log-model pairs from various domains, including healthcare, finance, and IT service management. The results demonstrate that REACH outperforms state-of-the-art conformance checking algorithms in terms of computation time and number of solved problems. Specifically, REACH solves 95% of the tested log-model pairs in under 45 seconds, whereas other state-of-the-art algorithms fail to align as many pairs even with a 300-second time limit.

In conclusion, the performance improvements achieved by this work enable efficient computation of optimal alignments for Petri net process models. By eliminating the need to rely on fast but less accurate methods, the proposed algorithm opens up new possibilities for the application of advanced conformance checking techniques in real-world scenarios.

DeclareAligner: A Leap Towards Efficient Optimal Alignments for Declarative Process Model Conformance Checking

Declarative process models offer a more flexible approach to modeling business processes, making them better suited for dynamic environments. Unlike procedural models, declarative models do not explicitly define the control flow, instead specifying *what* constraints must be satisfied during process execution without prescribing exactly *how* it should be done [26]. The flexibility of declarative process models introduces new challenges for conformance checking, as the search space of possible alignments becomes significantly larger.

The DECLARE language is the leading approach to declarative process modeling, offering a flexible and concise way to define constraints on business processes by filling constraint templates with relevant activities [43]. For instance, consider a hospital's patient discharge process, which includes activities such as medication prescription, patient discharge, and follow-up appointment scheduling. A DECLARE model can specify constraints for this process, like "a patient must receive medication before being discharged" or "a follow-up appointment must be scheduled after the discharge, without any other discharge in between". This allows for flexibility in the process while still enforcing key constraints.

When computing optimal alignments for declarative process models, existing approaches often struggle with scalability and efficiency. The state of the art [44, 45, 46] translates the DECLARE language into Linear-time Temporal Logic over finite traces (LTLf) formulas and then into finite-state automata [47]. These automata can then be used to compute alignments using algorithms similar to those used for procedural models. However, this translation process can lead to a loss of semantic simplicity and an increase in computational complexity. Specifically, the translation process can obscure the declarative nature of the original constraints, thereby forfeiting potential optimizations that could be leveraged if the declarative representation were used directly for computing optimal alignments.

To address this challenge, DECLAREALIGNER is proposed, a novel algorithm that extends the A* search algorithm to tackle the problem from a fresh perspective, leveraging the flexibility of declarative models by using them directly during the alignment process. The algorithm constructs a search space incrementally as it is explored, initializing with a starting state that embodies the entire trace. From this foundation, new states are iteratively generated through the application of targeted corrective actions, ultimately yielding a goal state that corresponds to a valid execution of the process. By comparing the original trace to a valid trace represented

by this goal state, an optimal alignment can be extracted, highlighting the deviations between the actual and expected behavior.

The proposed approach offers several significant innovations that enhance its performance and scalability. It only considers taking actions that directly contribute to repairing violated constraints, and proposes several search optimization strategies, including an heuristic that merges the suggested actions for all violated constraints of a state to provide accurate cost estimates. Additionally, states that cannot reach the optimal alignment are pruned early to eliminate unfruitful branches of the search space, and actions that are required by chain constraints and do not interfere with other constraints are applied before the search. These innovations enable DeclareAligner to efficiently compute optimal alignments for declarative process models, making it a valuable tool for conformance checking in dynamic business environments. The algorithm's native support for declarative models leveraging their flexibility and its optimization strategies make it well-suited for handling complex process models and large event logs.

The evaluation of DeclareAligner is conducted on a comprehensive dataset consisting of $8{,}054$ trace-model pairs, which includes real-life and synthetic data. A case study is presented to demonstrate the practical utility of the proposed algorithm. Furthermore, an ablation study is conducted to evaluate the impact of each component of DeclareAligner on its overall performance. The results show that Early Pruning, Constraint Preprocessing, and Grouped Fixes are all key components that contribute significantly to the algorithm's efficiency. Lastly, an in-depth benchmarking against the state of the art is performed. DeclareAligner achieves an average execution time of 7.6 seconds for the complete dataset, which is substantially lower than the next best algorithm (DeclareReplayer) with an average execution time of 72.5 seconds. Additionally, the proposed optimizations enable DeclareAligner to successfully compute alignments for 231 trace-model pairs that remain unsolved by all other algorithms within a 5-minute time limit.

In conclusion, DeclareAligner offers a novel approach to computing optimal alignments for declarative process models, leveraging the flexibility of these models and introducing several optimization strategies to improve performance and scalability. The algorithm's ability to efficiently compute optimal alignments makes it a valuable tool for conformance checking in dynamic business environments, and its evaluation demonstrates its superiority over existing state-of-the-art approaches.

Efficient Conformance Checking of Rich Data-Aware Declare Specifications

Previous sections focus on the control-flow aspect, providing a high-performance foundation for conformance checking within Process Mining. However, as processes become increasingly complex and data-driven, there is a growing need to extend conformance checking to data-aware declarative models. Declarative modeling is prized for its flexibility, which unfortunately expands the alignment search space and complicates conformance checking. Introducing event-level data further amplifies this complexity, as simultaneous consideration of control-flow and data perspectives becomes necessary. Despite these challenges, data-aware models yield richer, context-sensitive insights, thereby justifying the need for the increased computational effort.

The need for more expressive, multi-perspective constraints in process modeling is well-established [48, 49, 50]. Consider, for example, a shipping scenario governed by a DECLARE response constraint between the events "Package Shipment" (A) and "Delivery Confirmation" (B): whenever a shipment occurs, a corresponding confirmation must follow. However, the timing of this confirmation depends on additional conditions. If the package weighs less than 10.5 kg and is bound for a specific geographic region, the confirmation must be received within 3 days of shipment. Otherwise, the deadline extends to 10 days. An algorithm that considers only the control-flow perspective would be unable to enforce these nuanced temporal and data-aware rules, thereby overlooking critical aspects of the service-level guarantees embedded in the business process.

The Multi-Perspective DECLARE framework (MP-DECLARE) is the leading approach to data-aware declarative process modeling, offering a flexible and concise way to define constraints on business processes that integrate both control-flow and data aspects [51]. However, devising alignment-based conformance checking for declarative, data-enriched specifications remains largely unresolved. Prior approaches concentrate on control-flow only, using representations such as Dynamic Condition Response (DCR) graphs [52], DECLARE constraints [53], or finite-trace Linear Temporal Logic (LTLf) [54]. To the best of our knowledge, the only existing work on data-aware alignment using DECLARE is [28]. However, their approach is limited to numeric data and basic variable-to-constant comparisons —e.g., $x > 5$—, without support for comparisons between variables across different events—an important limitation that our work aims to overcome.

To address this challenge, Data-Aware DECLARE Aligner is proposed, a novel approach integrating two complementary strategies: an A* search mechanism for exploring the space of possible alignments and Satisfiability Modulo Theories (SMT) solving for handling control-flow and data constraints [55]. SMT is a powerful decision problem that incorporates various background theories, such as arithmetic and uninterpreted functions, to determine the satisfiability of logical formulas. This makes SMT a promising tool for computing data-aware alignments, particularly in complex domains where rich data types and intricate relationships between variables are involved.

Data-Aware DECLARE Aligner defines a search space whose initial state encodes the original trace as an SMT formula. Neighbors are generated by applying actions over an state that directly contribute to repairing a violated constraint of the process model. Each explored state undergoes SMT-based violation detection, and any discrepancy triggers the generation of child states via targeted repairs. The algorithm proceeds until a goal state with minimal repair cost and no violations is reached, from which the optimal alignment can be reconstructed. The correctness of the proposed method is proven, and extensive experiments demonstrate its efficiency, even with complex data conditions.

The proposed approach effectively handles complex data constraints by leveraging the power of SMT solving. It treats timestamps like any other data attribute, enabling simultaneous reasoning about both control-flow and data constraints. This unified treatment also facilitates the specification and enforcement of temporal correlation constraints. Data-Aware DECLARE Aligner has been optimized to improve performance, with several key strategies employed to reduce computational complexity. One key optimization involves selecting a violation to repair, where the algorithm chooses the violation resulting in the fewest child states, delaying state explosion and improving search efficiency. Additionally, dead-ends are detected by precomputing all violations and checking if any repairs are applicable; if not, the state is pruned from the search space, eliminating unfruitful branches. Furthermore, unsatisfiable states are identified and removed, as conflicting conditions were added during state generation.

The evaluation of the proposed approach demonstrates its efficiency. Experiments were performed on a dataset with varying levels of complexity. Initially, simple data conditions were used for constraint activations and targets to ensure compatibility with the only comparable algorithm [28]. On average, Data-Aware DECLARE Aligner aligns traces 2.9 times faster than the state-of-the-art method. Furthermore, the evaluation highlights the approach's capability to manage intricate data dependencies, such as correlation conditions, by extending the original

dataset with significantly more complex constraints.

In conclusion, the proposed approach offers a novel solution to the challenge of efficient conformance checking for rich data-aware declarative process models. By integrating A* search and SMT solving, the approach defines a search space that supports rich data types and complex data expressions, making it possible to handle intricate data conditions not supported by the state of the art.

Conclusions

This PhD has made significant contributions to the field of Process Mining by investigating the efficient computation of optimal alignments between event logs and process models. The primary motivation behind this research stems from the recognition that existing conformance checking techniques often struggle to efficiently compute optimal alignments for real-life processes, particularly in scenarios where event log traces are large or process models are complex.

The key findings of this research demonstrate the efficiency of the proposed algorithms in computing optimal alignments between event logs and procedural, declarative, and data-aware declarative process models. The results have significant implications for the field of Process Mining, enabling organizations to gain deeper insights into their processes, identify areas for improvement, and optimize their operations with greater confidence.

This work has introduced a comprehensive framework for efficient alignment-based conformance checking, combining formal foundations with complete implementations and extensive evaluation on real and synthetic datasets. The proposed algorithms have been shown to outperform existing state-of-the-art methods demonstrating significant improvements in efficiency and flexibility. Ultimately, this PhD has successfully achieved all of its objectives, paving the way for a new generation of Process Mining techniques and cementing a lasting impact on the field.

CHAPTER 1

INTRODUCTION

1.1 Motivation

In the vast and intricate landscape of modern organizations, processes play a pivotal role as they embody the operations that are systematically undertaken to achieve specific business objectives [1]. Processes are ubiquitous in various aspects of life, playing an important role in shaping the operations of organizations worldwide. They can be found in diverse fields such as IT service management [2], athletic performance optimization [3, 4], healthcare [5, 6, 7, 8], and software development [9, 10, 11], among others. Processes, in their most basic form, represent a series of coordinated activities designed to achieve a specific goal or set of goals, whether it be the production of goods, the delivery of services, or the execution of administrative functions [56]. These processes can range from simple, routine procedures to complex, interconnected networks of actions that involve numerous stakeholders, resources, and decision points.

The importance of processes extends beyond mere operational management to encompass strategic, tactical, and compliance dimensions as well [1]. For instance, understanding how processes are actually executed —as opposed to how they are theoretically designed— can reveal opportunities for cost reduction, quality enhancement, and risk mitigation. Moreover, in highly regulated industries, the ability to demonstrate process compliance through detailed event logs is an important factor for meeting legal and regulatory requirements, thereby avoiding potential penalties and reputational damage.

Given the significance of processes, it becomes clear why their analysis and improvement

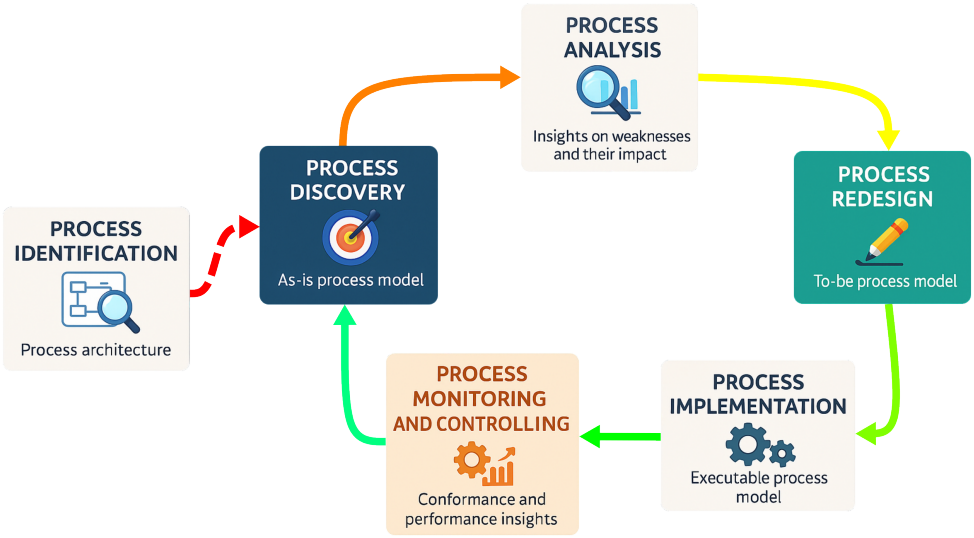


Figure 1.1: Business Process Management framework lifecycle (adapted from (13)).

have emerged as critical disciplines within the broader realm of Business Process Management (BPM) [12]. BPM encompasses a range of methodologies, techniques, and tools aimed at designing, executing, and continuously improving business processes. The BPM lifecycle is illustrated in Figure 1.1, which provides an overview of the key stages involved in managing business processes.

The BPM lifecycle is characterized by an iterative framework, comprising multiple stages that build upon one another to facilitate ongoing process improvement [57]. This cyclical structure is initiated by identifying the process, its underlying architecture, and relevant performance metrics. The current process state is then mapped and analyzed to uncover inefficiencies and areas for enhancement. Subsequent stages involve redesigning the process to address identified issues, implementing these changes, and ultimately monitoring the revamped process to assess its conformance. This continuous cycle of evaluation, refinement, and implementation enables organizations to incrementally improve their processes over time.

The traditional approach to managing business processes has been hindered by the need for manual intervention at various stages, including design, analysis, and improvement. This labor-intensive set of tasks can be particularly challenging when dealing with complex processes.

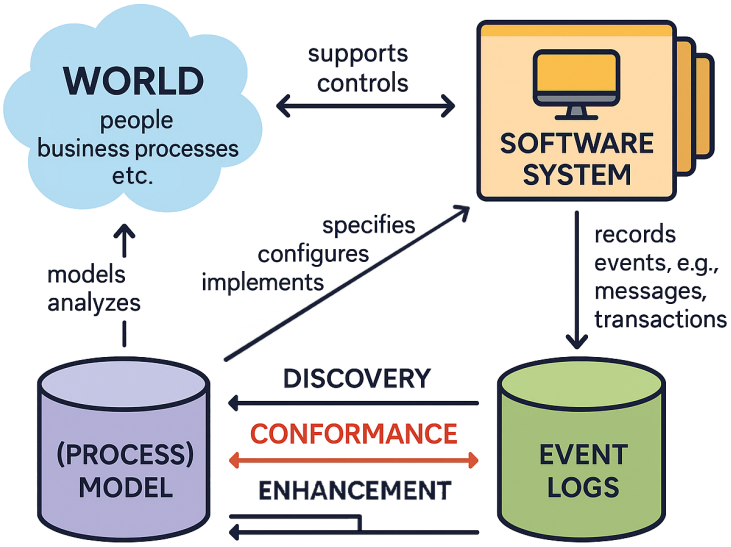


Figure 1.2: Process Mining framework (adapted from (15)).

To address this issue, a new field of research named Process Mining has emerged [14], seeking to leverage data-driven insights to automate and optimize BPM.

Process Mining harnesses the wealth of information captured by software systems that support organizational processes. As illustrated in Figure 1.2, this information is recorded as event logs, which serve as a digital footprint of an organization's operations. By analyzing them, Process Mining techniques can discover a process model that accurately reflects the current state of the process, check its conformance with new event logs to identify discrepancies and deviations, and ultimately enhance the process models using the knowledge gathered from previous analyses.

The key to unlocking this potential lies in the development of techniques that can extract valuable knowledge from event logs, which can then be used to analyze and improve business processes. By bridging the gap between data mining, machine learning, and business process management [58], it becomes possible to create a more seamless and efficient approach to process optimization. Ultimately, the goal of this emerging field is to provide organizations with the tools [59, 60] they need to streamline their operations, reduce inefficiencies, and

improve overall performance. By harnessing the power of data analytics and Process Mining, businesses can gain a deeper understanding of their processes and make targeted improvements that drive real results.

Process Mining complements BPM by using specialized algorithms and techniques to reveal, monitor, and improve processes by extracting knowledge from event logs [14]. As illustrated in Figure 1.2, Process Mining techniques are generally divided into three primary categories:

Discovery techniques aim to automatically construct a process model from event log data, without any prior knowledge of the process. The goal is to discover the underlying process structure, including the sequence of activities, decision points, and interactions between different process components. Lots of process model discovery techniques have been proposed or improved in recent years, such as the α++ Miner [61], the Split Miner [62], the Inductive Miner [63] and the Heuristics Miner [64], which have shown promising results in automatically constructing accurate and meaningful process models from event log data.

Conformance techniques [29] focus on comparing the observed behavior in the event log with a predefined process model, to identify deviations and inconsistencies. The goal is to check whether the actual process execution conforms to the expected or desired behavior, and to highlight and quantify any discrepancies. Conformance checking techniques include token-based replay [30, 65, 31] and alignments [66, 67, 41, 68, 38, 32, 39], among many others.

Enhancement techniques aim to improve the existing process model or event log data, by adding new insights or perspectives. The goal is to enhance the process by identifying opportunities for improvement, such as reducing cycle times, increasing throughput, or improving quality. Enhancement techniques include process repair [69, 70] and process extension by considering additional perspectives like the resources involved [71].

Conformance checking is a vital area within Process Mining, as it enables the alignment of actual behavior captured in event logs with the expected behavior outlined in process models [1]. This PhD research is centered on advancing conformance checking, with a particular focus on addressing the complexity and variability inherent to real-life business processes. Existing techniques often struggle to cope with the complexity of this kind of processes. To bridge

this gap, the goal of this work is to develop more scalable, flexible, and efficient methods for conformance checking. By tackling current limitations, this research aims to offer practical solutions that support organizations in better understanding, managing, and optimizing their processes for improved performance and competitiveness.

1.1.1 Event log

At the heart of Process Mining lies the event log, also simply called a log, a comprehensive record of events that occur during business process execution, capturing a digital footprint of actual behaviors, outcomes, and interactions within an organizational context [72]. Even when the recorded process execution data does not match the exact event log format described in this section, it can usually easily and quickly be preprocessed into this format, making it suitable for analyzing with Process Mining techniques.

Most applications focus on analyzing the order of activities within each case, known as the control-flow perspective. For this purpose, each event on these logs must contain at least activity names, execution times, and case identifiers [73]. Event data often encompasses a broader scope of information, including resource allocation, participant roles, cost, and other contextual elements [27]. By leveraging this richer dataset, advanced algorithms can provide more informed insights, ultimately supporting more effective data-driven decision-making. Algorithms that do not support the data perspective simply ignore all unsupported attributes of the event log.

Definition 1.1: Event

Let U_A be the set of all possible activities, U_C be the set of case identifiers, $U_{\mathcal{T}}$ be the time domain, and $\mathcal{D}_1, \ldots, \mathcal{D}_n$ be the domains of all attributes associated with each event, where $n \geq 0$. An event e is a tuple $(a, c, t, d_1, \ldots, d_n)$, where $a \in U_A$, $c \in U_C$, $t \in U_{\mathcal{T}}$, and $d_i \in \mathcal{D}_i \cup \{\perp\}$ for $i = 1, \ldots, n$, with $\perp$ representing the absence of a value. The activity a, timestamp t and case identifier c are the only required components of an event.

Definition 1.2: Trace

A trace $\sigma \in \mathcal{S}$ is a sequence of events $\sigma = \langle e_1, \ldots, e_m \rangle$, where for any two events $e_i = (a_{e_i}, c_{e_i}, t_{e_i}, \ldots), e_j = (a_{e_j}, c_{e_j}, t_{e_j}, \ldots) \in \sigma$ with $i, j \in [1, m]$, if $j > i$, then the case identifier of e_i is equal to the case identifier of e_j (i.e., $c_{e_i} = c_{e_j}$) and the timestamp of e_j is greater than or equal to the timestamp of e_i (i.e., $t_{e_j} \geq t_{e_i}$). The length of a trace σ is denoted by $|\sigma| = m$.

At its core, a trace embodies the chronological sequence of events that pertain to a specific case, providing a detailed record of the activities executed over time. In a simplified representation, where data attributes are not considered, a trace can be abstracted as a straightforward ordered sequence of activities, denoted as $\langle a_1, \ldots, a_m \rangle$.

Definition 1.3: Event log

An event log $L = [\sigma_1, \ldots, \sigma_n]$ is a multiset of traces, i.e., an unordered collection of traces where duplicates are allowed.

The significance of event logs lies in their ability to provide organizations with a data-driven means of gaining insights into process behavior, identifying inefficiencies, and uncovering opportunities for improvement [74]. By analyzing these logs, organizations can develop a deeper understanding of their actual process execution, compared to theoretical models, enabling meaningful process enhancements, operational efficiency optimization, and improved competitiveness.

Example 1.1: Event log

Table 1.1 shows an example of an event log taken from an e-learning process. Each row of the table describes a unique event of the log. In total, this log comprises 17 events, which are organized into 3 distinct traces as determined by the *Case* attribute. There are 4 different activities being performed throughout these events.

Notably, each event includes a resource attribute, specifying the platform utilized by the student to complete the activity. Moreover, events that perform the *Test* or *Exam* activities have an additional outcome attribute, which indicates whether the student passed or failed, providing valuable insight into their performance.

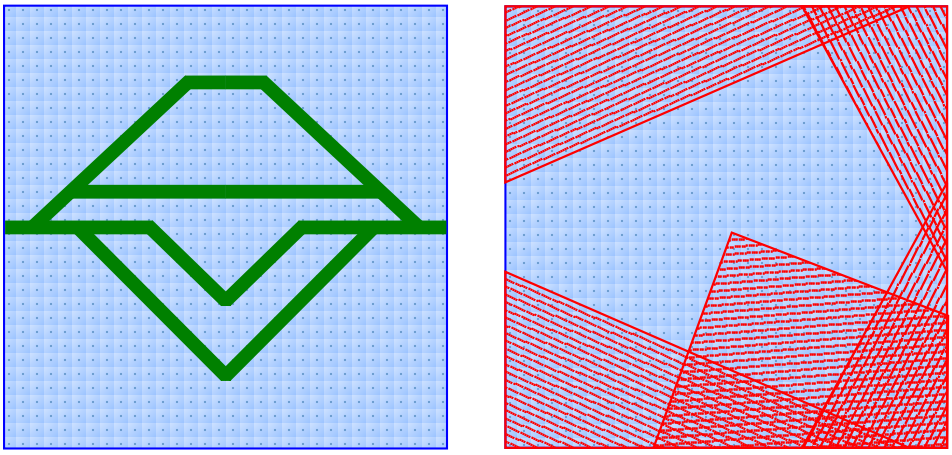
Table 1.1: Example event log from an e-learning process, comprising 17 events from 3 distinct traces that capture the actions of individual users as they progress through a course.

Case	Timestamp	Activity	Resource	Outcome
#234	2025-01-05 14:30:00	Enroll	Mobile App	
#987	2025-01-06 10:15:00	Enroll	Website	
#675	2025-01-10 09:45:00	Enroll	Mobile App	
#987	2025-01-12 11:50:00	Class	Virtual Classroom	
#234	2025-02-02 11:15:00	Class	Virtual Classroom	
#675	2025-02-05 14:10:00	Class	Virtual Classroom	
#987	2025-02-20 16:30:00	Test	Mobile App	Fail
#234	2025-03-15 20:10:00	Class	Virtual Classroom	
#675	2025-04-01 08:05:00	Class	Virtual Classroom	
#234	2025-04-22 17:40:00	Test	Assessment Portal	Pass
#675	2025-05-01 10:50:00	Class	Virtual Classroom	
#675	2025-05-06 12:20:00	Test	Mobile App	Pass
#987	2025-05-15 14:30:00	Class	Virtual Classroom	
#675	2025-05-25 15:50:00	Class	Virtual Classroom	
#675	2025-06-01 09:00:00	Exam	Assessment Portal	Pass
#987	2025-06-05 11:10:00	Exam	Assessment Portal	Fail
#234	2025-06-10 14:45:00	Exam	Assessment Portal	Pass
⋮	⋮	⋮	⋮	⋮

The event log reveals the actions undertaken by each student, illustrating their engagement with the e-learning platform over time. For instance, the student #234 performs events with the following activities: $\langle \textit{Enroll}, \textit{Class}, \textit{Class}, \textit{Test}, \textit{Exam} \rangle$.

1.1.2 Process model

Process models, also simply referred to as models, are essential for understanding and analyzing business processes. They distill the complexity of a process into a simplified yet meaningful abstraction, capturing its key activities and relationships. In essence, a process model describes the behavior of a set of valid traces, which are the finite sequences of events that can occur within the process.



(a) Procedural models define a set of acceptable behaviors highlighted in green thick lines.

(b) Declarative models define a set of forbidden behaviors highlighted in red thin lines.

Figure 1.3: Intuitive differences between procedural and declarative process models. The blue dotted square represents the universe of events over which the process models define allowed traces (adapted from (25)).

Definition 1.4: Process model

Let U_E be the universe of events and $U_{\mathcal{M}}$ be the universe of process models. A process model $\mathcal{M} \in U_{\mathcal{M}}$ accepts a set of traces defined by its language $\mathcal{L}(\mathcal{M}) \subseteq U_E^*$. The notation U_E^* refers to the Kleene star operator, which represents the set of all possible finite sequences of any length over the events in U_E.

The language of a model $\mathcal{L}(\mathcal{M})$ describes the patterns of events that are admissible by the process model $\mathcal{M}$, creating a formal contract for describing the model's behavior.

There are various formalisms used to represent process models, including Business Process Model Notation (BPMN) [17], Petri nets and workflow nets [18], DECLARE [19], Directly-Follows Graphs (DFGs) [20], Process Trees [21], Causal nets [22] or Dynamic Condition Response Graphs (DCR Graphs) [23].

Process modeling techniques can be categorized into two primary approaches [16]: procedural and declarative. Figure 1.3 illustrates the fundamental differences between these

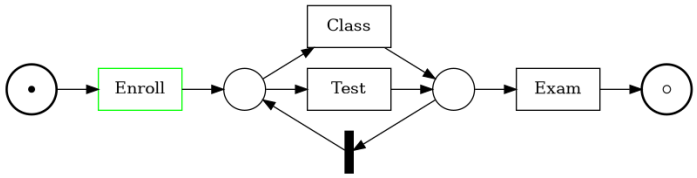
approaches. Procedural models, which explicitly outline all allowed sequences of events, are juxtaposed with declarative models, which permit all behaviors by default and only impose constraints as necessary. This contrast is visually represented in the figure, where procedural models define a comprehensive set of acceptable behaviors (Figure 1.3(a)), whereas declarative models delineate a set of forbidden behaviors (Figure 1.3(b)). This visualization highlights the distinct focuses of each approach, with procedural models prescribing exactly *how* a process should be executed and declarative models specifying *what* constraints must be satisfied during process execution [26].

This PhD research aims to leverage the benefits of both procedural and declarative process modeling approaches, recognizing that each has its own strengths. Notably, it also incorporates the data perspective into declarative process modeling while improving upon existing limitations of the state of the art. The specific modeling approaches underpinning the proposed framework are introduced next.

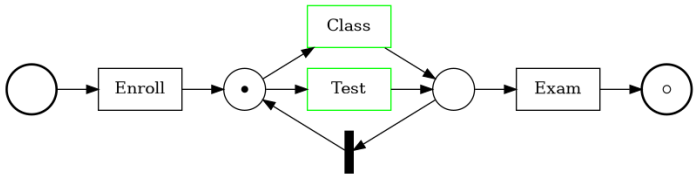
Procedural process models: Petri nets and workflow nets

Petri nets [24] are the most widely used procedural modeling language in the field of Process Mining, owing to their rigorous mathematical formalization that enables them to effectively describe and analyze complex distributed systems. A Petri net is a directed bipartite graph consisting of two types of nodes: places and transitions. Places represent the states or conditions that a process can be in, while transitions represent the activities or actions that can be performed. The directed arcs connecting places and transitions define the flow of the process, indicating which activities can be executed in sequence.

To analyze the runtime behavior of a Petri net, the concept of tokens is introduced. Tokens represent the current state of the process and are moved from one place to another as transitions are fired, allowing the progress of the process to be tracked. A marking refers to the distribution of tokens over the places in the Petri net, providing a snapshot of the current state. The initial marking is a special case, representing the starting point of the process, and is typically defined by placing a token in each of the designated initial places. Finally, the Petri net also includes a final place, which, when reached with tokens, signals the successful end of the process.



(a) Initial marking.



(b) Marking after executing *Enroll*.

Figure 1.4: Two markings of a Petri net modeling the running example of an e-learning process.

Example 1.2: Petri net

An example of a Petri net is shown in Figure 1.4(a), which models the simplified process of the e-learning course introduced in Example 1.1. Places are represented by circles and transitions are shown as rectangles. A filled rectangle is a silent transition, and other transitions have a label that matches the activity name it represents. An enabled transition, i.e., a transition that can be executed in the current state, is highlighted with a green border. Both the initial and final places are highlighted with a thicker border, and the final place contains a small inner circle.

In a Petri net, each transition has a set of input places and a set of output places, which are connected to the transition via directed arcs. A transition is said to be enabled when all its input places contain at least as many tokens as there are arcs pointing from the place to the transition. When a transition is enabled, it can be fired or executed, which has the effect of consuming the tokens from the input places and producing new tokens in the output places. The execution of a transition, therefore, involves the removal of tokens from the input places and the addition of tokens to the output places, as indicated by the incoming and outgoing arcs respectively.

Example 1.3: Petri net replay

The Petri net shown in Figure 1.4(a) has a token on the initial place. This enables the *Enroll* transition since the place connected by its only input arc has a token. Executing this enabled transition generates a token in its output place, resulting in the marking shown in Figure 1.4(b).

This enables the *Class* and *Test* transitions and disables the *Enroll* transition. At this point, the *Class* and *Test* transitions can be executed any number of times, but at least one of them is executed once. Lastly, the *Exam* transition must be fired to complete the process by reaching the final place.

Workflow nets [24], a special class of Petri nets, are particularly well-suited for modeling business processes. A workflow net is defined as a Petri net with a single initial place, a single final place, and a structure that ensures every part of the net is reachable from every other part. This type of net that will be used in the context of this research. Example 1.2 is also a workflow net as it satisfies all the previous requirements.

One issue with using Petri nets to model flexible and dynamic environments is that the resulting models can become overly complex. This complexity arises from the need to specify all valid execution paths at each point in the process, leading to process models such as the one shown in Figure 1.5, often called Spaghetti models [76]. This can make it difficult to understand and analyze the process behavior, limiting the effectiveness of Petri nets in these scenarios.

Declarative process models: DECLARE

Declarative process models offer a promising solution for modeling and analyzing dynamic and flexible business processes, where traditional procedural models like Petri nets may fall short. By focusing on the declaration of constraints and rules that govern the process behavior, rather than explicitly defining its control flow, declarative models provide a more flexible and adaptable approach to process modeling.

Declarative process models are particularly well-suited for environments where the process is subject to frequent changes, or where the execution paths are highly variable. They allow for a more modular and incremental approach to process modeling, where new constraints and rules can be easily added or removed as needed. This makes them an attractive option

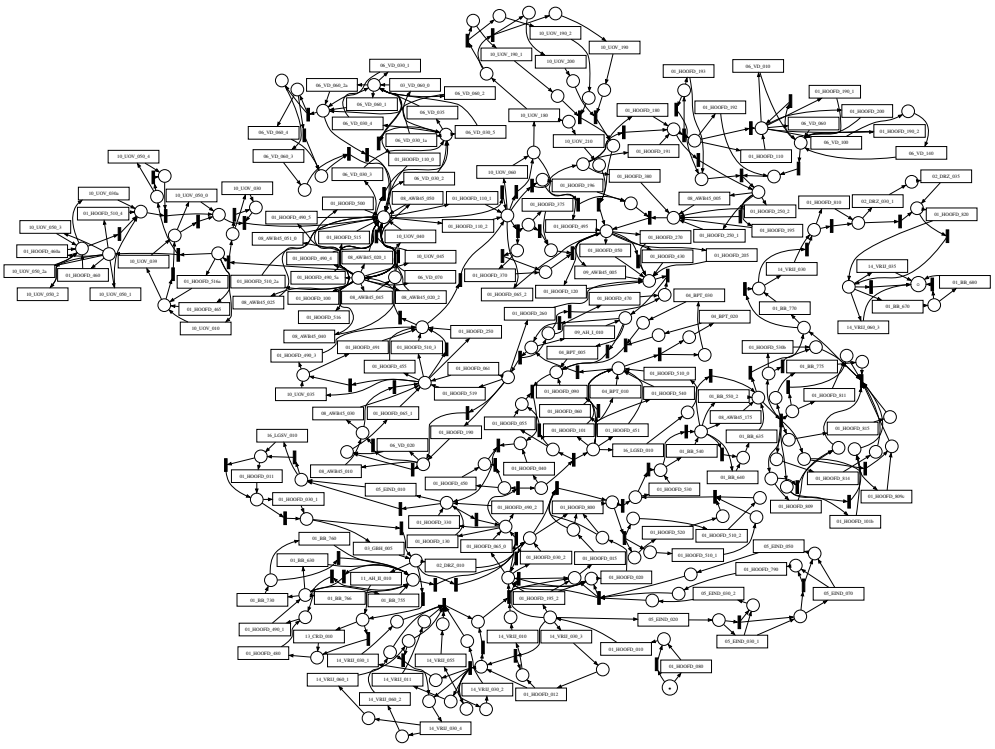
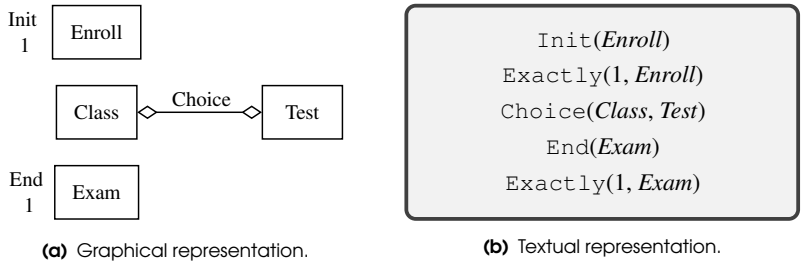


Figure 1.5: Example of a complex process model discovered from an environmental permit application process (75).

for organizations that need to respond quickly to changing market conditions or customer requirements.

Among declarative process models, DECLARE has emerged as a dominant paradigm [25]. The popularity of DECLARE can be attributed to its simplicity, expressiveness, and flexibility. By using a set of predefined templates [77], such as `Existence`, `Response`, and `NotChainSuccession`, DECLARE allows modelers to define the rules by simply filling in the parameters of those templates and adding them to the model as needed.



(a) Graphical representation.

(b) Textual representation.

Figure 1.6: A DECLARE model for the running example of an e-learning process.

Example 1.4: DECLARE

The DECLARE model presented in Figure 1.6 describes the same process as the procedural model illustrated in Figure 1.4. The constraints employed in this model are:

- `Init`(*Enroll*): the process must start with the *Enroll* activity.
- `Exactly`(1, *Enroll*): the *Enroll* activity must occur exactly once.
- `Choice`(*Class*, *Test*): This constraint stipulates that either the *Test* or *Class* activity (or both) must be executed at some point during the process. Notably, there are no additional constraints governing the frequency or order of these activities.
- `End`(*Exam*): The process must end with the *Exam* activity.
- `Exactly`(1, *Exam*): The *Exam* activity must occur exactly once.

Some constraints in DECLARE have an activation condition, which determines when the constraint becomes relevant. For example, the `Response`(*A*, *B*) constraint is activated when activity *A* appears in the trace, and it then checks if activity *B* —the target activity— eventually appears in the remainder of the trace. Constraints without an activation condition, such as `Init`(*A*), are always considered active. If the condition enforced by the constraint is met, the constraint is satisfied; otherwise, it is violated. Constraints that are never activated are considered vacuously satisfied.

Definition 1.5: DECLARE model

A DECLARE model $\mathcal{M}$ defines a language $\mathcal{L}(\mathcal{M})$ consisting of all possible traces that satisfy all declared constraints: $\sigma \in \mathcal{L}(\mathcal{M})$ if and only if $\forall c \in \mathcal{M} : satisfies(\sigma, c)$, where $\mathcal{M}$ is the DECLARE model and $satisfies(\sigma, c)$ denotes that σ satisfies the constraint c. This ensures that only traces that meet all the specified constraints are accepted by the model.

Data-aware declarative process models

The integration of the data perspective into process models is necessary for capturing the intricate relationships between processes and data. Both procedural and declarative process modeling approaches can be extended to incorporate this perspective, although the current state of the art in declarative process modeling lags behind that of procedural modeling, particularly in terms of integrating the data perspective. This gap can be attributed to the relative novelty of declarative approaches, which have only recently gained significant attention in the field of process modeling. Consequently, this research focuses on developing data-aware declarative process models.

To incorporate the data perspective, event logs are leveraged to extract relevant attributes associated with each event. These attributes can encompass various aspects, including time-related information —e.g., activity durations, times between activities—, organizational context —e.g., resources involved, roles, cost—, or other contextual information stored in the event log. In data-aware declarative models, data conditions are defined to enhance constraints with additional requirements based on these extracted attributes.

In the context of data-aware DECLARE [28], data conditions can be attached to either activation or target activities in data-aware declarative models. Activation conditions impose an extra requirement that must be met for a constraint to be considered active, whereas target conditions introduce an additional requirement that must be fulfilled for an activated constraint to be deemed satisfied. Notably, when a constraint evaluates the target condition, it has already been activated and thus has access to the activation event data (if available). Target conditions that utilize activation data are specifically referred to as correlation conditions.

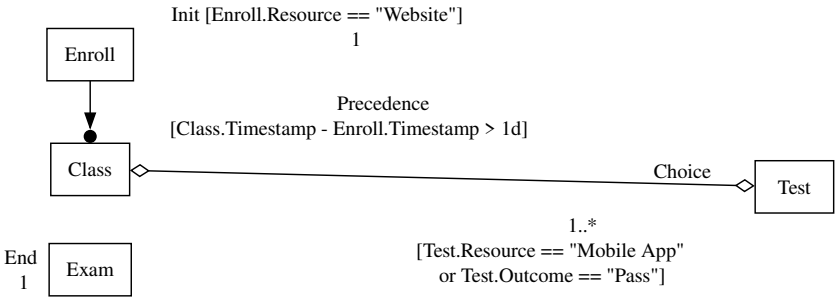


Figure 1.7: A data-aware DECLARE model for the running example of an e-learning process.

Example 1.5: Data-aware DECLARE

Figure 1.7 shows some data-aware constraints that extend the DECLARE process model of the running example (Figure 1.6). The modified or added constraints are:

- `Init`(*Enroll*) | $Enroll.Resource ==$ "Website" |: The process must start with an enrollment activity performed through the website.
- `Precedence`(*Enroll*, *Class*) |||$Class.Timestamp - Enroll.Timestamp > 1d$|: If the student attends a class, they must have enrolled at least 24 hours earlier.
- `Existence`(1, *Test*) | $Test.Resource ==$ "Mobile App" or $Test.Outcome ==$ "Pass" |: Students must take at least one test using the mobile app or pass at least one test.

Satisfiability Modulo Theories (SMT) [55] extends the classical Boolean satisfiability problem (SAT) by incorporating various background theories, such as arithmetic, bit-vectors, arrays, and uninterpreted functions. By leveraging these theories, SMT solvers can determine the satisfiability of logical formulas expressed in first-order logic. The versatility of SMT makes it an attractive solution for data-aware conformance checking [50], particularly in complex domains where rich data types and intricate relationships between variables are involved.

1.1.3 Conformance checking and optimal alignments

Conformance checking is a key component of Process Mining, which receives a discovered or manually-designed process models and an event log, and outputs key insights required for

process enhancement or data-driven decision-making.

Within conformance checking, several techniques have been proposed to analyze the degree of conformance between an event log and a process model. One popular approach involves executing each event of the trace in the process model while recording all execution errors. For Petri nets, this technique is called token-based replay [78] and records the missing and forgotten tokens. Similarly, for declarative models like DECLARE, all constraint activations, satisfactions, and violations can be reported. This results in very fast conformance checking approaches as they only need to simulate the trace over the model.

However, these replay-based techniques have significant problems. The diagnostics they provide are often hard to understand for non-experts because they are related to the specific model used. Furthermore, the errors they find may not be directly related to the root cause of the issue, making it harder to identify and fix the real problem.

Example 1.6: Misleading diagnostics of replay-based conformance checking

To illustrate this, consider a Bakery process where the baker wants to make a cake. Suppose the baker mistakenly inserts the ingredients for a muffin instead of a cake, but then follows the recipe for the cake perfectly. A replay-based approach would enter the branch of the process that manages making a muffin successfully, but later report errors when activities like *Mix Cake Batter* or *Apply Cake Frosting* are performed.

This diagnostic information is misleading as it suggests that the problem lies with the later activities, rather than the much more likely single-activity failure of selecting incorrect ingredients. A more insightful analysis based on alignments would reveal that the root cause of the conformance issue is the incorrect ingredients used at the beginning of the process.

Alignment-based approaches have emerged as a widely adopted technique for conformance checking due to their ability to provide more accurate and insightful diagnostics [29]. An alignment is a sequence of moves that reconciles the events in a trace with a valid execution of the process model. By doing so, alignments establish a connection between the actual behavior observed in the trace and a feasible path through the process model, even if the original trace cannot be directly executed. This capability has significant benefits, including facilitating process repair and enhancement [79], as well as providing more reliable data for techniques that benefit from conforming traces, such as predictive monitoring algorithms [80, 81].

The different types of moves that can comprise an alignment are:

- Synchronous moves: execute an event in the trace and the corresponding model activity.
- Log moves: advance to the next event in the trace without executing a model activity.
- Model moves: execute model activity without advancing to the next event in the trace.

Any process model and trace can be trivially aligned by performing a sequence of log moves on all trace events and model moves along a valid execution path of the process model. However, such alignments are of little practical value, as they fail to highlight meaningful deviations between the trace and the model. To overcome this, a cost function is introduced that assigns a numerical cost to each move. The standard cost function typically assigns a cost of 0 to synchronous moves —where a trace event corresponds directly to a model activity— and a cost of 1 to all asynchronous moves, which include both log moves and model moves. The total cost of an alignment is then computed as the sum of the costs of all its constituent moves.

An optimal alignment is one that achieves the minimum possible cost among all valid alignments for a given trace and model. Optimal alignments are especially valuable because they identify the minimal and most plausible set of deviations needed to reconcile the observed behavior with the process model. This minimization problem, although computationally intensive, leads to diagnostic outputs that are both accurate and actionable, helping analysts understand precisely where and how the behavior diverged.

Example 1.7: Optimal alignment for the running example

To illustrate the concept of alignments, consider the example shown in Table 1.2. This figure depicts two optimal alignments of cost 2 for the trace $\langle \textit{Enroll}, \textit{Exam}, \textit{Test} \rangle$ and the process model from the running example. Each column represents a move, indicating the activity of the trace in the first row and the executed transition of the model in the second row. Moves are executed from left to right to take the trace and the model from the initial state to the end. The symbol » is used for asynchronous moves, indicating that no action is taken.

The only difference between the two alignments is the second move: in the first, there is a model move inserting *Class*, whereas in the second, there is a model move inserting *Test* instead. These moves indicate that the trace skipped a required activity, which could be either *Class* or *Test*. Both alignments end with a log move skipping the last *Test* event in the trace, which indicates that this event should not have occurred.

Table 1.2: Two optimal alignments for the control-flow-only trace ⟨*Enroll, Exam, Test*⟩ and the model represented procedurally in Figure 1.4 and declaratively in Figure 1.6.

Log:	Enroll	»	Exam	Test
Model:	Enroll	Class	Exam	»

Log:	Enroll	»	Exam	Test
Model:	Enroll	Test	Exam	»

In traditional conformance checking, the focus lies primarily on the control-flow perspective, i.e., whether the sequence of recorded events in the trace matches the structure of the process model. However, real-world processes often involve more than just the ordering of activities: events typically carry additional information in the form of data attributes which may be subject to data conditions defined by the process model.

Data-aware conformance checking introduces new challenges: even if the sequence of activities in a trace appears valid at the control-flow level, it may still violate data conditions specified in the process model. For instance, a process model may require that an activity like *Enroll* is only valid if performed through the "Website", but the trace might indicate it was performed using the "Mobile app". This kind of mismatch cannot be detected or resolved by control-flow alignments alone.

To address data mismatches in data-aware alignments, a new type of asynchronous move is introduced: the edit move. An edit move functions similarly to a synchronous move, in that it matches a trace event to a corresponding model activity. However, unlike synchronous moves, it allows for one or more attribute values of the event to be modified in order to satisfy the data conditions specified by the process model. Additionally, in a data-aware alignment, model moves must also provide attribute values that comply with the data conditions defined by the process model.

Allowing arbitrary edit moves would defeat the purpose of data-aware alignments. Hence, analogously to other asynchronous moves, edit moves are assigned a cost. The standard cost function assigns a cost equal to the number of edited attributes in the move. This way, the alignment process still favors minimal deviation: whenever possible, the alignment algorithm will prefer exact (synchronous) matches and will only apply edit moves when no other lower-cost alternative exists.

By integrating edit moves into the alignment framework, it is possible to obtain optimal alignments that reflect both control-flow and data deviations. This enriched alignment approach enables analysts to precisely identify not only where the process behavior diverges but also

Table 1.3: Optimal alignment for the trace #234 from Table 1.1 and the data-aware DECLARE model from Figure 1.7. Only edited attributes are shown.

Log:	Enroll{Resource="Mobile App"}	Class	Class	Test	Exam
Model:	Enroll{Resource="Website"}	Class	Class	Test	Exam

which specific data attributes contribute to these deviations. Consequently, data-aware optimal alignments provide a more comprehensive basis for process analysis and improvement. However, this also makes the alignment computation more complex, as the search must now consider combinations of control-flow and data deviations to find the alignment with the lowest total cost.

Example 1.8: Data-aware optimal alignment for the running example

To further illustrate the concept of data-aware alignments, consider the example shown in Table 1.3. In this scenario, a student enrolled in the e-learning course using the mobile app. However, the constraint $\texttt{Init}(\textit{Enroll}) \mid \textit{Enroll}.\textit{Resource} == \text{“Website”} \mid$ requires the first event of the trace to be an enrollment performed via the website. Since this condition is not met, the most cost-effective way to align the trace with the model is to modify the data attribute of the event. This leads to the application of an edit move, as shown in the optimal alignment.

Note that this modified attribute value must still satisfy any other conditions that interact with it throughout the process model. Fortunately, all other data conditions defined in the process model are already satisfied by the trace, eliminating the need for additional asynchronous moves.

The need to compute optimal alignments efficiently

The computation of optimal alignments in conformance checking requires an efficient search algorithm to explore the potentially infinite number of possible alignments between a process model and a trace from an event log. Some approaches convert the alignment problem to the Planning Domain Definition Language and leverage a classical automated planner to compute optimal alignments [42, 54, 37, 28, 46]. However, these methods often encounter scalability

issues due to the necessary conversion to a planning task and the lack of control over the way that the automated planner tackles the alignment problem.

The A* search algorithm [33] has emerged as a prominent choice for this task due to its renowned properties of optimality and completeness. As an optimal algorithm, A* is guaranteed to find the shortest path to the goal state, which in the context of conformance checking, corresponds to the optimal alignment between the process model and the event log. Furthermore, the completeness of A* ensures that it will always find a solution if one exists, making it a reliable choice for computing alignments. Its ability to incrementally define the search space and the use of a heuristic function to efficiently navigate the search space makes it a widely adopted algorithm for conformance checking. Moreover, the flexibility of A* allows for the design of new and more efficient search spaces, which can be achieved through techniques such as state pruning and problem reformulation.

One key aspect of A* search is the utilization of heuristics to guide the search towards the most promising states, thereby optimizing the exploration process. While simple heuristics, such as those based on the number of remaining events in the trace [32], can facilitate faster exploration, they may also result in a larger number of explored states. In contrast, complex heuristics, including those rooted in Integer Linear Programming (ILP) [39, 40, 41], can effectively reduce the number of states to be explored, albeit at the cost of increased processing time per state. The use of ILP-based heuristics is particularly effective in detecting specific types of errors, such as event swaps between the start and end of the trace; however, it can lead to prolonged execution times for other cases. Therefore, developing a heuristic that strikes a balance between simplicity and accuracy is valuable, as it would minimize the time required for computation while reducing the number of states to be explored.

Suboptimal techniques have also been proposed as a compromise between result quality and computational cost, aiming to provide approximate alignments at a reduced computational expense. These approaches include decomposing models into smaller parts [82, 74, 83] and utilizing evolutionary algorithms [84]. A significant drawback of these suboptimal techniques is that the accuracy of their approximations is generally unknown, which can be a major concern in applications where precise conformance checking is crucial. For instance, in scenarios where the stakes are high, such as in financial or healthcare institutions, relying on potentially inaccurate alignments can have serious consequences, making it essential to prioritize optimal alignment techniques despite their higher computational costs.

The growing complexity of process models has led to a significant increase in the size of the

search space, resulting in performance degradation of existing algorithms. Moreover, optimal alignments can be beneficial for various downstream applications that rely on conforming logs, such as training predictive monitoring algorithms [80, 81, 85], which can benefit from using alignments to find the closest path through the process model instead of training using traces that are not accepted by the process model. Similarly, detecting concept drift [86] in processes can be more effectively achieved with optimal alignments, as they enable a more nuanced understanding of deviations from the expected behavior, rather than simply reporting an excessive number of errors. Furthermore, process enhancement and repair [79] techniques also rely on accurate alignments to identify areas for improvement and propose corrective actions. Efficiently computing optimal alignments therefore unlocks more potential of these applications and ongoing research in this area is necessary to address the challenges posed by increasingly complex process models and large event logs.

The research focus of this PhD is situated at the forefront of this demanding yet rewarding area of investigation. By exploring novel methodologies and techniques for efficiently aligning event logs with procedural, declarative and data-aware declarative process models, this work aims to significantly improve the performance and scalability of conformance checking algorithms, enabling faster and more accurate analysis of large and complex event logs found in a wide range of domains, from healthcare and finance to manufacturing and logistics.

1.2 Hypothesis and objectives

The efficient alignment of event logs with process models is an important challenge in the field of Process Mining. Despite the existence of various conformance checking techniques, the computation of optimal alignments remains a significant obstacle, particularly when dealing with large and complex process models that are characteristic of many modern organizations. The complexity of these models, combined with the sheer volume of event data, can render traditional conformance checking methods computationally expensive. In fact, for many real-world cases, computing optimal alignments is not feasible with reasonable resources, rendering traditional conformance checking methods ineffective.

The field of Process Mining is rapidly advancing, yet it is hindered by scalability and reliability issues in alignment-based conformance checking. To overcome these challenges, innovative solutions are needed. A* search [33] has proven its worth in various optimization problems, but to reach its true potential in computing optimal alignments, it may require the

development of sophisticated heuristics, efficient search space pruning methods and, potentially, a fundamental redesign of the alignment search problem. The incorporation of complementary optimization techniques, such as SMT, also holds promise for enhancing efficiency, particularly when considering the data perspective. In light of these challenges and opportunities, the hypothesis guiding this PhD thesis is:

Hypothesis

Exploring various optimization techniques, including advanced A* heuristics, novel search space pruning strategies, and innovative problem reformulations, can significantly improve the efficiency of computing optimal alignments, thereby enabling the computation of complex alignments in real-life processes that are currently intractable with state-of-the-art methods.

To investigate the stated hypothesis, it is necessary to define a set of research objectives that, if achieved, could demonstrate the feasibility of significantly improving the efficiency and scalability of computing optimal alignments. These objectives aim to address key challenges across different process modeling paradigms and, collectively, serve as the foundation for validating the broader research claim. Specifically, the following objectives are proposed:

O1. Develop a high-performance method for computing optimal alignments in procedural process models. This objective aims to leverage the A* search paradigm, enriched with advanced heuristics and state-space pruning techniques, to efficiently compute optimal alignments between event logs and Petri nets. Promising strategies involve designing heuristics that quickly pinpoint necessary future activities, reducing the search space by identifying unnecessary moves, and enabling efficient Petri net execution.

O2. Create a scalable approach for computing optimal alignments in declarative process models. The goal is to develop a method to efficiently compute optimal alignments between event logs and DECLARE process models. The proposed search strategy could focus on repairing constraint violations rather than constructing alignments move by move. Promising directions include heuristics that consolidate multiple repair suggestions into a single cost estimate, as well as optimization techniques that reduce the number of states to explore by detecting dead-ends, preprocessing the input problem and performing multiple actions at the same time.

O3. Advance the computation of optimal alignments in data-aware declarative process models with flexible data constraints. This objective focuses on supporting the optimal alignment of event logs and data-aware DECLARE process models in the presence of complex data conditions by combining A* search with SMT solving. Promising techniques and enhancements include encoding control-flow and data constraints into unified logical formulations, treating timestamps as regular data attributes to enable temporal reasoning, and supporting expressive data conditions such as correlation conditions while maintaining efficiency.

1.3 Methodology

This research follows a systematic and iterative methodology, grounded in the scientific method. In terms of the specific steps involved, the methodology can be outlined as follows:

Objective definition. The process begins with a clear definition of the research objectives, which involves identifying the problem, establishing the research goals, and anticipating the limitations of the potential solution. This initial phase is crucial in providing a solid foundation for the subsequent stages.

Analysis of the related work. Following the objective definition, a comprehensive analysis of the state of the art is conducted. This stage involves an exhaustive review of existing solutions, techniques, and approaches relevant to the research objectives. The primary aim is to identify gaps in current knowledge, understand the strengths and weaknesses of available methods, and pinpoint areas that require further investigation.

Development. With the insights gained from the state-of-the-art analysis, the development of a solution commences. This stage encompasses the formulation of theoretical frameworks, design of algorithms, and implementation of systems. An incremental approach is employed for software development, where features are implemented and tested in an iterative manner. This methodology allows for flexibility and adaptability, enabling adjustments to be made as needed.

Validation. The validation of the proposed solution is a critical phase, involving rigorous testing and benchmarking to ensure its effectiveness and reliability. The performance of the solution is quantified and compared to existing methods, with the results informing

potential modifications or refinements. If the performance does not meet expectations, the process reverts to previous stages, allowing for revisions and improvements to be made.

This methodology allows a flexible and non-linear research process. Different stages may overlap, and retrospective revisits are permitted to augment or improve the results. This adaptability enables the research to respond to emerging findings, adjust to changing circumstances, and ultimately produce a more robust and reliable solution that contributes meaningfully to the existing body of knowledge.

1.4 Contributions

This thesis introduces a comprehensive framework for efficient alignment-based conformance checking across procedural, declarative, and data-aware declarative process models. Each contribution addresses the corresponding objective stated in Section 1.2, combining formal foundations with extensive evaluation on real and synthetic datasets. The results demonstrate clear improvements over the state of the art in both accuracy and performance. The main contributions of this thesis are summarized as follows:

C1. Efficient optimal alignment algorithm for procedural process models. This research contributes an innovative algorithm for computing optimal alignments between event logs and Petri net models, leveraging A* search and optimization techniques to significantly improve performance and scalability. The resulting algorithm enables the alignment of large and complex logs and models, previously infeasible with existing methods. The main contributions of this algorithm include:

- A novel heuristic that efficiently identifies required activities and compares them with the remaining events of the trace.
- Techniques to reduce the number of states explored by the A* search, including mandatory move detection and a greedy cost estimation approach.
- An efficient execution method for Petri nets using a partial reachability graph to improve performance.

C2. Novel algorithm for optimal alignments in declarative process models. This research introduces a novel optimal alignment algorithm tailored for DECLARE process models, designed to efficiently resolve constraint violations while ensuring optimality through targeted optimization strategies. The algorithm departs from traditional move-by-move alignment construction and instead adopts a violation-centric approach, where alignments are computed by systematically identifying and repairing constraint violations. The main innovations that underpin this contribution include:

- A novel search space that focuses on performing actions that contribute to repairing violated constraints.
- A heuristic function based on estimating the minimal number of constraint repairs required.
- The early elimination of dead-end states to reduce the effective search space.
- The preprocessing of constraints that allow the search to start from a more advantageous initial state.
- The application of multiple repairs in a single action, reducing intermediate steps.

C3. Advanced conformance checking for data-aware declarative process models. This research pushes forward the boundaries of conformance checking by introducing Data-Aware DECLARE Aligner, an efficient algorithm designed to compute optimal alignments while handling complex data conditions. By integrating A* search with SMT solving, the approach achieves both high efficiency and expressiveness. The main features of this contribution include:

- A hybrid approach where A* search directs the exploration, while SMT formulas handle state representation, violation detection, and repair action generation.
- The inclusion of timestamps within SMT encodings, enabling temporal reasoning alongside other data attributes.
- Support for expressive data constraints, including correlation constraints that extend beyond the capabilities of previous methods.

1.4.1 Journals

Publication

Jacobo Casas-Ramos[a], Manuel Mucientes[a], and Manuel Lama[a]. REACH: Researching Efficient Alignment-based Conformance Checking. *Expert Systems with Applications*, 241:122467, 2024.
DOI: 10.1016/j.eswa.2023.122467

Journal Impact Factor (JCR 2024): 7.5

- Computer Science, Artificial Intelligence: **Q1** (28/204)

[a]CiTIUS, Universidade de Santiago de Compostela, Santiago de Compostela, Spain.

Publication

Jacobo Casas-Ramos[a], Manuel Lama[a], and Manuel Mucientes[a]. DeclareAligner: A Leap Towards Efficient Optimal Alignments for Declarative Process Model Conformance Checking. *Engineering Applications of Artificial Intelligence*, 2025.
DOI: 10.1016/j.engappai.2025.111683

Journal Impact Factor (JCR 2024): 8.0

- Computer Science, Artificial Intelligence: **Q1** (25/204)

[a]CiTIUS, Universidade de Santiago de Compostela, Santiago de Compostela, Spain.

1.4.2 International Conferences

Publication

Jacobo Casas-Ramos[a], Sarah Winkler[b], Alessandro Gianola[c], Marco Montali[b], Manuel Mucientes[a], and Manuel Lama[a]. Efficient conformance checking of rich data-aware declare specifications. In *Business Process Management*, pages 88–105, Cham, 2026. Springer Nature Switzerland.
DOI: 10.1007/978-3-032-02867-9_7

Conference ranking (ICORE, CORE2023): A

[a]CiTIUS, Universidade de Santiago de Compostela, Santiago de Compostela, Spain.
[b]Free University of Bozen-Bolzano, Bolzano, Italy.
[c]INESC-ID/Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal.

1.5 Dissertation structure

This research is structured to provide a comprehensive overview of the research conducted on efficient alignment-based conformance checking for procedural, declarative, and data-aware declarative process models. The following chapters outline the contents of this thesis:

- Resumo and Summary provide a summary in both Galician and English, respectively.
- Chapter 1 introduces the research motivation, objectives and contributions, outlining the methodology used to achieve efficient alignment-based conformance checking for procedural, declarative, and data-aware declarative process models. This chapter sets the stage for the contributions made by this dissertation, providing a clear understanding of the research context and goals.
- Chapter 2 presents the contribution **C1**: an efficient algorithm for computing optimal alignments in procedural process models, addressing objective **O1** by leveraging A* search with an advanced heuristic and other optimization techniques to improve performance and scalability. The algorithm is evaluated using a comprehensive set of experiments, demonstrating its performance in aligning large and complex logs and models.
- Chapter 3 introduces the contribution **C2**: a novel approach for declarative process model conformance checking, targeting objective **O2** by utilizing A* search and heuristic techniques to quickly compute optimal alignments and enable efficient analysis for declarative process models. This approach is shown to outperform existing methods, providing a significant improvement in computational efficiency.
- Chapter 4 presents contribution **C3**: an efficient conformance checking of rich data-aware declarative process models, focusing on objective **O3** by combining A* search and SMT solving to handle advanced data conditions and achieve both high-efficiency and enhanced expressiveness. The resulting algorithm is capable of handling complex data dependencies, making it a valuable tool for real-world process analysis applications.

- Chapter 5 summarizes the main findings and contributions, highlighting the key results and outcomes of the research conducted. This chapter provides an overview of the accomplishments of this dissertation, including the development of efficient algorithms for alignment-based conformance checking and their evaluation using comprehensive experiments. Furthermore, possible paths for future research are identified.
- Appendix A formalizes the semantics of data-aware DECLARE constraints employed in this research. It provides precise definitions that underpin the interpretation of traces with respect to complex data conditions, forming the theoretical foundation for data-aware conformance checking.
- Appendix B presents the proofs and supplementary material associated with Data-Aware DECLARE Aligner, as introduced in Chapter 4. This chapter includes formal results, assumptions, and technical details that substantiate the correctness, completeness, and practical applicability of the proposed algorithm.
- Appendix C enumerates the scientific publications that form the basis of this dissertation. For each publication, this chapter specifies the individual contributions of the doctoral candidate and details regarding copyright and licensing.

Chapter 2

REACH: Researching Efficient Alignment-based Conformance Checking

The contents of this chapter are partially extracted from the following publication:

Publication

Jacobo Casas-Ramos[a], Manuel Mucientes[a], and Manuel Lama[a]. REACH: Researching Efficient Alignment-based Conformance Checking. *Expert Systems with Applications*, 241:122467, 2024.
DOI: 10.1016/j.eswa.2023.122467

Journal Impact Factor (JCR 2024): 7.5

- Computer Science, Artificial Intelligence: **Q1** (28/204)

[a]CiTIUS, Universidade de Santiago de Compostela, Santiago de Compostela, Spain.

As discussed in Chapter 1, the importance of Process Mining in understanding and improving business processes cannot be overstated. One important area of Process Mining is conformance checking [1], which involves comparing the actual execution of a process with its intended model to identify deviations and opportunities for improvement. Conformance checking techniques are important for ensuring that business processes operate efficiently and

effectively, and alignment-based approaches have emerged as the most successful solutions for this purpose [87].

Within the realm of conformance checking, procedural process models, particularly Petri nets, have gained significant attention due to their ability to accurately represent complex business processes. Some methods employ token-based replay techniques [30, 31] for conformance checking. Although these approaches can be computationally efficient, they lack the capability for root cause detection, which limits the quality of insights they can provide. As a result, despite their potential speed advantages, token-based replay techniques may not offer the same level of diagnostic information as alignment-based methods.

However, computing optimal alignments between event logs and Petri net models remains a challenging task, especially when dealing with large and complex models. The difficulty lies in the computational complexity required to find these alignments, which can result in extended processing times and even prevent the computation of some alignments. Despite the existence of several alignment-based conformance checking approaches for Petri nets [32, 37, 38, 39, 40, 41, 42], there is still room for improvement in terms of performance as they often struggle with complex models and extensive logs.

This chapter presents REACH an efficient algorithm which leverages A* search and optimization techniques to improve performance and scalability in computing optimal alignments between event logs and Petri net models. REACH incorporates a series of optimizations to enhance performance and scalability. They significantly reduce the number of states that need to be explored to reach the optimal solution and the processing time spent on each state. The main contributions of the proposal are:

1. A new heuristic that quickly finds required activities —activities that must be executed to reach the end of the model— and compares them with the remaining trace in order to provide more accurate estimates and speed up the algorithm without losing its optimality.

2. Techniques for reducing the number of states explored by the A* algorithm. These include optimizations that check and force the execution of required moves by exploring the rest of the trace and the model. Furthermore, a greedy algorithm that returns a sub-optimal alignment is used as an upper bound of cost for the main algorithm. These optimizations filter states that will not lead to the optimal alignment, as another state will lead to an alignment of less cost. This also removes all neighbors generated by the ignored states recursively, greatly reducing the computational cost.

3. Efficient execution of process models for alignment computation using a partial and incremental reachability graph. It works by saving information about how process models run for alignment computation. It prevents performing repeated model operations at each step of the A algorithm.

The approach is compared against a wide range of existing conformance checking algorithms for Petri net. The performance of REACH is evaluated using a comprehensive set of experiments, involving 227 pairs of logs and models. The results show that REACH outperforms other state-of-the-art proposals in terms of runtime and alignment quality, making it a valuable tool for conformance checking.

The remainder of this chapter is organized as follows. Section 2.1 studies and compares previous work. Next, Section 2.2 defines all the required concepts on which this work is based. The algorithm is described in detail in Section 2.3. Section 2.4 describes the experiments and discusses the results, compared to the current state of the art. Finally, the conclusions and the future work are presented in Section 2.5.

2.1 Related work

Conformance checking is a very active field in Process Mining. One of its most popular approaches is token-based replay over Petri nets [30, 31]. It involves executing each event of the Petri net as they appear in the trace. If an event cannot be executed, the tokens required to execute it are inserted and counted. Once the full trace has been replayed, all the remaining tokens that are not in a sink place are counted. The fitness metric, which is a measure of conformance between the log and the model, is computed based on these counts. Unfortunately, this technique is less accurate than alignment-based approaches, as it assumes the model is always correct. Furthermore, the provided diagnostics are hard to understand for the end-user, as they are tied to the Petri net model representation and replay. In [88] they propose a decomposition-based extension of token-based replay that is more efficient, but it shares the aforementioned disadvantages.

A recent development in this field is stochastic conformance checking [89], which involves comparing event logs and process models while recognizing that logs represent only a subset of possible behaviors. The major disadvantage of the approach is that stochastic process model discovery is a necessary prerequisite for applying this technique. It is a computationally

expensive task that has much higher memory requirements than traditional process discovery techniques. The algorithm depends on an input parameter (probability mass) that makes the returned metrics more stable when it is increased. However, the runtime of the algorithm increases very quickly with respect to this parameter for most datasets. This is even worse if the model presents concurrency and looping behavior.

Alignment-based techniques [32] can identify and explicitly list all discrepancies, enabling the detection of optimal trace executions through the model. These techniques still rely internally on the execution of process models, but they can find the path through the model with the least number of discrepancies possible. There are also techniques, like behavioral alignments [90], that provide textual summaries of conformance without actually computing alignments. These descriptions might be easier for end-users, but they are not as useful as alignments. Alignments link event data to the model, and can be used for post-processing tasks such as fitness calculation, performance analysis, model repair, or prediction tasks.

Alignments are considered the standard for conformance checking. In order to find the optimal alignments, most approaches use a pathfinding algorithm like A* for aligning the trace and the model. A* requires a heuristic, which defines the way to explore the state space of the problem. On the one hand, simple heuristics lead to faster exploration but more explored states [32, 37, 38], which makes computing alignments for medium or large models impractical without other states reduction techniques. On the other hand, complex heuristics focus on reducing the number of states at the cost of more processing time per state. An illustrative instance of a complex heuristic is rooted in Integer Linear Programming (ILP), utilizing constraints extracted from the model, the remaining trace, and an optimization cost function. Given a state (marking in the model and remaining trace events), the ILP solver is capable of determining the minimum cost that a solution may have, so it is suitable as a heuristic for A* [39, 40, 41]. However, a drawback of complex heuristics is that the computational cost for each discovered state often surpasses the efficiency gained through state reduction.

Leoni et al. [42] convert the alignment problem to the Planning Domain Definition Language and use an external automated planner to compute the alignments. Their algorithm can modify the planning framework for alignment computation. Nevertheless, their approach depends on the blind A* heuristic, which guarantees optimality but significantly underestimates the remaining cost.

Considering the exponential complexity of optimal algorithms, researchers have introduced non-optimal techniques for computing alignments. These methods were proposed as a

compromise between the quality of the results and the computational cost. These techniques do not guarantee that the alignment with the least cost will be found, but they provide approximations with a quality that depends on the model and the log. One such technique is [84], utilizing an evolutionary algorithm to offer improved alignment approximations, albeit without the assurance of discovering all optimal alignments. Another approach that uses local search to reduce processing time and memory usage is [91], with the added limitation of only being able to return one of the alignments. It is worth mentioning [92], which also performs an iterative search in order to find a possibly non-optimal alignment.

A prevalent method for non-optimal alignments involves decomposing models into smaller, more computationally efficient parts and aggregating partial results, even though this may not always result in optimal alignments [82, 74, 83]. Alternatively, a recomposing technique that is capable of obtaining optimal alignments from the partial pseudo-alignments was developed in [67], but it was only executed with manual decompositions of models. Another decomposing optimal technique is described in [93], but it is limited to sound and safe workflow nets.

Finally, another type of non-optimal technique is based on building automata capable of aligning the log and the model [68, 94, 66, 95]. These approaches, while non-optimal, give good approximations of the optimal alignments for most cases. The method in [94] splits the activities into multiple subsets to handle very complex models with many activities. Consequently, it increases performance on models with lots of activities at the expense of lower cost accuracy of the resulting alignments.

The state of the art still struggles when facing complex models and logs, leading to large runtimes, and even not being able to compute some alignments. To address this issue, several proposals have relaxed restrictions and returned non-optimal alignments. However, in this chapter, we present an A*-based algorithm that increases performance while still providing optimal alignments. This algorithm introduces several techniques for reducing the number of states of the search space to be explored: a new heuristics to better guide the states to explore, a greedy algorithm to find an upper cost bound of the optimal alignment, and two optimizations that check each state to force certain mandatory moves instead of generating all possible neighboring states. Furthermore, our approach introduces an efficient way to execute process models that relies on a partial and incremental reachability graph in order to speed up the computation required for each state. These techniques speed up computation and mitigate the scalability problem currently present in the state-of-the-art proposals.

Table 2.1: Example of a log corresponding to an e-learning process excluding data. A gray font highlights the events of the trace *Case234*.

Timestamp	Trace ID	Activity
2021-09-01 22:16:29	Case234	Enroll
2021-09-03 16:12:24	Case675	Enroll
2021-10-04 08:09:56	Case234	Class
2021-10-19 00:00:13	Case234	Class
2021-11-01 04:10:07	Case675	Class
2021-11-15 02:09:48	Case234	Test
2021-11-29 13:53:30	Case675	Test
2021-12-13 07:24:41	Case675	Class
2022-01-17 23:03:31	Case675	Exam
2022-01-19 06:42:33	Case234	Exam
...		

2.2 Preliminaries

To perform conformance checking a log and a process model are needed. Logs consist of events, organized into traces, with each trace representing the execution of the associated process model. Every event must have the execution timestamp, the trace identifier that groups the events of the same process execution, and the executed activity. Definitions 1.1 to 1.3 formalize these concepts.

For the sake of simplicity, we define traces as a sequence of executed activities, sorted by the execution timestamps of each event or in the order of appearance in the log in case of ties. Furthermore, the $+\!\!+$ operator concatenates two sequences, and, given a trace σ, the notation $\sigma[i{:}]$ refers to the subsequence from position i (zero-indexed, inclusive) to the end of the sequence.

Example 2.1: Log without data attributes

Table 2.1 shows an example of a log with two traces from an e-learning platform, where each row of the table is an event. Traces in this context refer to the actions of a student

in a virtual course. To track the actions of an individual student, we can extract a trace by filtering the recorded events (log table rows) with the same trace identifier. A student executed the activity in the column *Activity* at the moment indicated by the column *Timestamp*. For instance, the trace identified as *Case234* documents the sequence of activities executed by the student: $\langle Enroll, Class, Class, Test, Exam \rangle$.

A process model describes the allowed behavior by giving activities a structure with initial and final states. In the realm of conformance checking, Petri nets are the dominant technique for representing process models.

Definition 2.1: Petri net

A Petri net is a tuple $PN = (P, T, F, \lambda)$ where

- P is the set of places.
- T is the set of transitions. These may contain silent transitions τ, which are related to the model structure, and non-silent transitions, which are related to the execution of activities. T_s denotes the set of silent transitions and $T_{ns} = T \setminus T_s$ is the set of non-silent transitions.
- $P \cap T = \emptyset$.
- $F \in (P \times T) \cup (T \times P)$ is the set of directed arcs that connect places to transitions and vice versa.
- $\lambda : T_{ns} \rightarrow A$ maps every non-silent transition to a label —an activity that may appear in the log. Multiple transitions can have the same label, as our algorithm supports handling duplicate activities in the process model. We also use λ_r to perform the reverse mapping —map an activity to the set of non-silent transitions with the given label.

Petri nets are directed bipartite graphs where nodes are places and transitions. We denote $\bullet t$ as the input places ($\bullet t = \{p \in P \mid (p,t) \in F\}$) and $t\bullet$ as the output places ($t\bullet = \{p \in P \mid (t,p) \in F\}$) of a transition $t \in T$. These are the places directly linked by arcs to and from transition t, respectively. The same operator can be applied to places to retrieve the connected transitions. In order to execute Petri nets, it is necessary to introduce the concepts of token and marking. A place $p \in P$ can store any number of tokens in the marking M, as given by the function $tokens(M, p)$. Hence, we define a marking as a multiset of places that

represents the number of tokens in each place.

Definition 2.2: Marking

Let $PN = (P, T, F, \lambda)$ be a Petri net and let $\mathcal{B}$ be the power multiset function. A marking $M \in \mathcal{B}(P)$ of that Petri net is a multiset of places. This multiset represents the places that contain tokens and the number of tokens they contain. $M_0 \in \mathcal{B}(P)$ is the initial marking of the Petri net.

During the execution of Petri nets, tokens are added to and removed from places. Specifically, at the start of the process, the Petri net is initialized with the tokens of the initial marking M_0. A transition $t \in T$ is enabled at a marking M iff $\forall p \in {\bullet t} : tokens(M, p) > 0$. This condition means that the transition is enabled when there is at least one token available for consumption in each place connected to the transition through an input arc. To execute that transition t, all input places $p \in {\bullet t}$ consume one token; and all output places $p \in t{\bullet}$ receive a token. When a marking including tokens in a place marked as final is reached, the process is considered as finished.

Example 2.2: Petri net

Figure 1.4 shows an example of a Petri net that models a very simplified process of an e-learning course. Places are represented as circles and transitions as rectangles. A silent transition is shown as a thin black rectangle. The filled circles inside some places indicate the number of tokens they contain. The end place is shown with a circumference inside. Enabled transitions are shown with green borders. In this process, students can enroll in the course and, once enrolled, they can attend any number of classes and/or take tests, performing at least one of those activities. Lastly, a final exam is required to complete the course. In this example, the *Enroll* transition is executed, leading the Petri net from its initial marking (a) to the next marking (b). Due to the execution of *Enroll*, a token is consumed in the initial place, and one is produced in its output place, enabling the *Class* and *Test* transitions. A complete process execution for this model would be $\langle$*Enroll*, *Class*, *Exam*$\rangle$.

Workflow nets [96] are a class of Petri nets focused on modeling processes: they have a single input place and a single output place, and all transitions are in a path from the start to the

end places.

Definition 2.3: Workflow net

Let $PN = (P, T, F, \lambda)$ be a Petri net. It is a workflow net if and only if:

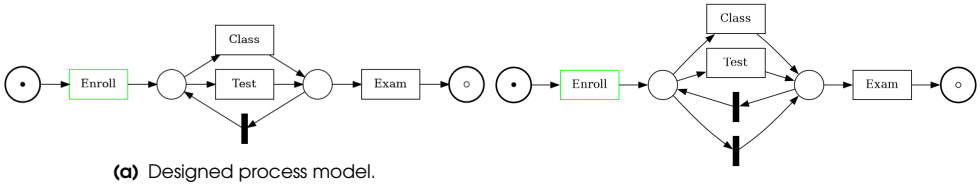
- There is a single input place $p_i \in P$. $\bullet p_i = \emptyset$.
- There is a single output place $p_o \in P$. $p_o \bullet = \emptyset$.
- Adding a transition t with $\bullet t = p_o$ and $t \bullet = p_i$, would cause the Petri net to become a strongly connected graph.

A workflow net has proper completion if for any reachable marking M by executing any sequence of enabled transitions $\langle [!\, t]_0, \ldots, t_n \rangle$, $tokens(M, p_o) > 0 \implies M = [p_o]$, i.e., any sequence of fired transitions that reach the end will only have one token in the output place.

In practice, process executions frequently deviate from the intended process models. For instance, in the context of the virtual course from Figure 2.1(a), students are expected to attend classes before taking the final exam. However, in reality, some students may choose to skip these lessons and still manage to complete the course, even though this behavior is not accounted for in the designed process model. This is shown in Figure 2.1. Alignments provide the necessary information to identify deviations of traces from the process model. An alignment can be computed for each trace given a process model, as a trace is an execution of the model. An alignment defines a path that traverses both the trace and the process model from start to end. It provides detailed conformance information useful for multiple tasks like model repair, auditing and prediction, even if the trace and the model do not match perfectly. This is achieved by associating the executed activities in the trace with the transitions in the model. Alignments are built from legal moves, which consume an activity of the trace and execute a non-silent transition with a matching label in the process model, or perform an asynchronous move by only executing a transition on the model (model move) or advancing on the trace (log move).

Definition 2.4: Legal move (Petri net)

Let M be the current marking in the model, let σ be the trace to align, and let i be the first index —zero-based— of the trace that still needs to be aligned so that $\sigma[i]$ is the



(a) Designed process model.

(b) Process model discovered from observed behavior.

Figure 2.1: Comparison between the designed process model of the running example (a) and the actual process model discovered from observed behavior (b).

next activity to align. Furthermore, let $M[t\rangle$ denote that marking M enables transition t. A legal move is one of the following.

- A synchronous move $(\sigma[i], t)$ is available iff $M[t\rangle$ and $\lambda(t) = \sigma[i]$. This move will update i with the next index of the trace. It will also update M by executing the transition t, i.e., $M - {}^\bullet t + t^\bullet$. Note that if the end of the trace was reached ($i = |\sigma|$), no more synchronous moves can be made.
- A log move $(\sigma[i],\gg)$ is available iff $i < |\sigma|$. This will increase i by one, advancing on the trace.
- A model move $(\gg, t)$ is available iff $M[t\rangle$. As a result of this move, M is updated to $M - {}^\bullet t + t^\bullet$. If t is a silent transition τ, the move is called a silent move.

Log and model moves are referred to as asynchronous moves.

Definition 2.5: Alignment (Petri net)

An alignment $\gamma = \langle \gamma_1, \ldots, \gamma_n \rangle$ is a sequence of legal moves (Definition 2.4). A complete alignment's moves $\gamma_1, \ldots, \gamma_n$ advance through the model and the trace. Alignments start from the initial marking of the model and the start of the trace and reach the end of the model and the end of the trace.

Definition 2.6: Cost function, alignment cost and optimal alignment (Petri net)

A cost function $C : (A \cup \{\gg\}) \times (T \cup \{\gg\}) \to (0, \infty)$ assigns a cost to each possible legal move, where A is the set of activities in the log and T is the set of transitions in

Table 2.2: Two optimal alignments using the default cost function for the trace ⟨*Enroll, Exam, Test*⟩ and the model in Figure 2.1(a).

Log:	Enroll	»	Exam	Test
Model:	Enroll	Class	Exam	»

Log:	Enroll	»	Exam	Test
Model:	Enroll	Test	Exam	»

the process model. Generally, the cost for synchronous moves is lower than the cost for asynchronous moves. Each valid alignment is assigned a cost $c_\gamma = c_{\gamma_1} + \cdots + c_{\gamma_n}$ which is the sum of all the costs of its moves. Hence, the optimal alignment for a given trace and model is the one with the minimum cost. There may be several different optimal alignments.

When conducting conformance checking, it is frequently beneficial to provide quality metrics. These metrics offer a concise summary of the results, condensing all calculated alignments into a single, easily interpretable value. One of the most well-established metrics is fitness. It gives an idea of the similarity between the log and the model based on the cost of the computed alignment. Fitness can be calculated for individual alignments or for the entire log. All optimal alignments of a trace have the same fitness as it is derived from the cost of the alignment.

Definition 2.7: Alignment fitness (Petri net)

Let γ be an alignment between the trace σ and the model M, with cost c_γ (Definition 2.6). The fitness for that alignment is defined as:

$$f_\gamma = 1 - \frac{c_\gamma}{\sigma_c + M_c}$$

where σ_c is the sum of costs for the asynchronous log moves for all of the trace activities and M_c is the sum of costs of the asynchronous model moves required for the minimum cost path through the model or, in other words, the cost of the optimal alignment of the empty trace and the model.

Example 2.3: Optimal alignments and fitness (Petri net)

Table 2.2 shows two alignments for the model depicted in Figure 2.1(a). Each column represents a move, indicating the activity of the trace in the first row and the executed transition of the model in the second row. Moves are executed from left to right to take the trace and the model from the initial state to the end. » is used for asynchronous moves indicating that no action is taken. A move in the model uses » in the trace row to show that only the transition in the model is executed, and a move in the log uses » in the model row. These alignments show the user how there was a skipped required activity that could have been either *Class* or *Test*, and the executed activity *Test* should not have been executed after the final *Exam* was taken, according to the model. The cost for each of these alignments, assuming a cost function that assigns a cost of 1 to asynchronous moves and 0 to synchronous moves, is 2 in both cases, and their fitness is $f = 1 - \frac{2}{3+3} = 0.\overline{6}$.

Fitness indicates how much the trace matches the model, quantifying all differences. Thus, it compares the cost of the given alignment with the cost of the alignment with only asynchronous moves: it first traverses the whole model via its shortest path adding the executed transitions as model moves, and then adds log moves for the whole trace. A fitness of 1 (perfect fitness) shows that the trace was executed correctly for the model (fitting trace), while lower fitness values indicate that discrepancies between the trace and the model were found. To calculate fitness for an entire log, the fitness values of individual traces are averaged. This gives an idea at a glance of how well the log conforms to the process model.

Definition 2.8: Log fitness

Let $\gamma_L = [\gamma_1, \ldots, \gamma_n]$ be a multiset of optimal alignments between each trace of the log $L = [\sigma_1, \ldots, \sigma_n]$ and the model M. The fitness of the log (f) is the average fitness of the alignments: $f = \left(\sum_{i=1}^{n} f_{\gamma_i}\right)/n$, where f_{γ_i} is the fitness of the alignment γ_i (Definition 2.7).

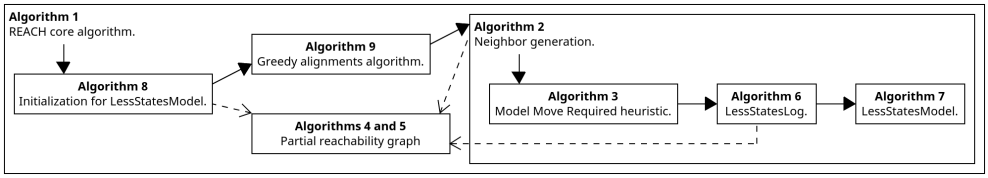


Figure 2.2: High-level diagram of REACH algorithms and their interactions. Continuous arrows show the control flow of the algorithm, while discontinuous ones show the usage of another algorithm, returning the control flow to the caller.

2.3 Conformance checking based on alignments

The goal of the REACH algorithm is to find the best alignment or all the best alignments for a model PN and each trace in a log L. Figure 2.2 shows an overview diagram containing the algorithms detailed in this section and how they relate to each other. Alg. 2.1 receives the inputs and runs the preprocessing steps of Algs. 2.8 and 2.9. Subsequently, the main search loop commences, iteratively invoking Alg. 2.2 to create neighbors of previously discovered states. Algs. 2.3, 2.6 and 2.7 are optimizations applied each time a neighbor is generated. Some algorithms depend on the partial reachability graph for the execution of operations on the model, as indicated by the discontinuous arrows of Figure 2.2.

Prior to initiating the processing of each log trace, the algorithm performs some preprocessing tasks. Firstly, once per model, it computes the shortest path through the model following the cost function (Alg. 2.1:3). While this is not required to find the optimal alignments, it is done to properly compute the fitness metric. Once each optimal alignment is computed, its fitness can be quickly calculated following Definition 2.7. Secondly, the log is simplified, detecting and counting duplicate traces, so that only one of those is processed. This task is executed only once per log (Alg. 2.1:4). Lastly, an optimization for reducing the number of states generated that will be explained in Alg. 2.8 is initialized.

A* is a complete and optimal search algorithm [33]. This implies that, if the problem has a solution, the algorithm will discover it —complete search algorithm—, and it will always discover the optimal solution if it exists. It stands out for its optimal efficiency with the main drawback being its exponential space complexity. It requires an admissible and consistent heuristic function to select the best path to pursue while guaranteeing optimality. This algorithm fits very well the alignment problem. A directed graph where the nodes are partial alignments

is defined, for which each available legal move (Definition 2.4) generates a new connected neighbor. The start node is the empty partial alignment, which contains no moves. Within this state graph, the algorithm needs to find the minimum cost path between the start and goal nodes, where a goal node is a complete alignment —an alignment that reaches the end of the model and the end of the trace. This is the kind of problem that A* solves with optimal efficiency, meaning that no other optimal algorithm would find the solution expanding fewer nodes if provided the same information.

The proposed conformance checking approach (Alg. 2.1) is grounded in the A* algorithm [32], a method for navigating the state space to identify the optimal alignment. Each state is a (partially) built alignment, meaning an alignment whose sequence of legal moves begins with the initial marking and the start of the trace. Concretely, states are defined as $S = (M, i, \gamma, c, h)$, where:

- M represents the state of execution of the process model. For Petri nets, it is the current marking.
- i is the zero-based index of the trace from which the remaining activities are not yet aligned.
- $\gamma = \langle \gamma_0, \ldots, \gamma_{i-1} \rangle$ is the partial or complete alignment, consisting of the sequence of legal moves taken.
- c is the current cost, which is the sum of the costs of the movements made.
- h is the heuristic value, i.e., an optimistic estimate of the remaining cost to complete the alignment.

Before executing the main algorithm, a greedy search is performed (Alg. 2.1:15-16). If it finds an alignment, it is used as a maximum cost limit to avoid generating unnecessary states and speed up the algorithm. The greedy search will be discussed further in Section 2.3.2. The algorithm begins with the initial state (Alg. 2.1:17), which is an alignment without any move, at the initial marking of the model and at the first event of the trace, with cost 0. States are kept in a priority queue that sorts states by increasing order of $S.c + S.h$ (Alg. 2.1:19), where $S.c$ is the cost and $S.h$ is the heuristic. $S.c + S.h$ is used to guide the exploration of A*, ensuring optimal results. The main loop (Alg. 2.1:21) explores each state in the order given by the priority queue while a solution is not found. The algorithm can also retrieve the set of all optimal alignments by iterating while $S.c + S.h$ is lower than the cost of any found solution ($min(\{S.c \mid S \in alignments\} \cup \{+inf\})$) plus the minimum cost of an asynchronous move (min_c) —as A* requires synchronous moves to have a negligible cost strictly greater than

Algorithm 2.1: REACH core algorithm.

Input: The process model PN, the log L to align to PN, and a boolean opt that is true to return all optimal alignments, otherwise one of them is given.

Output: Optimal alignments.

```
1:  procedure EXECUTE(PN, L, opt)
2:      results ← ∅
3:      PN.leastCost ← SHORTESTPATH(PN)              ▷ Needed for computing fitness
4:      L ← SIMPLIFYLOG(L)
5:      LESSSTATESMODELINIT(PN)                                          ▷ Alg. 2.8
6:      for all σ ∈ L do
7:          alignments ←ALIGN(PN, σ, opt)                          ▷ Align the trace
8:          results ← AGGREGATE(results, alignments)               ▷ Aggregate the results
9:      end for
10:     return results
11: procedure SHORTESTPATH(PN)
12:     S ←ALIGN(PN, EMPTYTRACE, false)[0] ▷ Align the empty trace to get the final state
13:     return S.c                          ▷ Return the cost c of the optimal alignment
14: procedure ALIGN(PN, σ, opt)
15:     greedyAlignment ←ALIGNGREEDY(PN, σ)                              ▷ Alg. 2.9
16:     maxCost ← greedyAlignment.c      ▷ The cost of the greedy alignment is the limit
17:     S ←INITIALSTATE(PN, σ)
18:     open ← ∅                                   ▷ Priority queue that stores states
19:     ADD(open, S)                               ▷ Insert state, sorted by S.c + S.h
20:     als ← ∅                                                ▷ Found optimal alignments
21:     while als = ∅ || (opt && S.c + S.h < min({S.c | S ∈ als} ∪ {+inf}) + min_c) do
22:         S ← POLL(open)                                     ▷ Extract the next best state
23:         if ¬ISFINAL(S) then
24:             ADDNEIGHBORS(σ, S, open, maxCost)                            ▷ Alg. 2.2
25:         else
26:             als ← als ∪ S          ▷ Record the final state of the optimal alignment
27:     end while
28:     return als
```

0. At each step, a state is removed from the priority queue (Alg. 2.1:22). If the given state is not final, the algorithm creates the neighbors of that state by using all the feasible legal moves (Definition 2.4). For each neighbor, it is necessary to update the process model state, the trace progress, the cost, and the heuristic. Otherwise, if the state is final, it is an optimal alignment and it is added to the alignments set (Alg. 2.1:23-26).

Algorithm 2.2: Neighbor generation.

Input: A current or parent state S and a priority queue $open$ for the new states to be inserted into.

Output: None, it adds neighbors to $open$.

```
1:  procedure AddNeighbors(σ, S, open, maxCost)
2:      etrs ← EnabledTransitions(S.M)  ▷ Alg. 2.4: enabled transitions from the S state
3:      if S.i < |σ| then                ▷ If the end of σ was not reached yet
4:          activity ← σ[S.i]            ▷ The next activity recorded in the trace
5:          trs ← λ_r(activity)          ▷ An activity may map to multiple model transitions
6:          for all tr ∈ trs ∩ etrs do
7:              AddMove(σ, S, open, Sync, tr, maxCost)   ▷ Generate synchronous moves
8:          end for
9:          if ¬LessStatesLog(σ, S) then                 ▷ Alg. 2.6
10:             AddMove(σ, S, open, Log, activity, maxCost)   ▷ Generate log movements
11:     if ¬LessStatesModel(σ, S) then                   ▷ Alg. 2.7
12:         for all tr ∈ etrs do
13:             AddMove(σ, S, open, Model, tr, maxCost)   ▷ Generate model movements
14:         end for
15: procedure AddMove(σ, S_parent, open, type, tr, maxCost)
16:     S ← NewState()
17:     if type = Sync then   ▷ Record the performed legal move in γ for the current state
18:         S.γ ← S_parent.γ ⧺ (λ(tr), tr)
19:     if type = Log then
20:         S.γ ← S_parent.γ ⧺ (tr,»)
21:     if type = Model then
22:         S.γ ← S_parent.γ ⧺ (», tr)
23:     if type ≠ Model then   ▷ Synchronous or log movements advance on the trace
24:         S.i ← S_parent.i + 1
25:     if type ≠ Log then   ▷ Synchronous or model movements advance on the model
26:         S.M ← ExecuteTransition(S_parent.M, tr)   ▷ Alg. 2.5
27:     if type ≠ Sync then S.c ← S_parent.c + 1 ▷ Update cost and heuristic for the new state
28:     else S.c ← S_parent.c + epsilon
29:     S.h = Heuristic(σ, S)                          ▷ Alg. 2.3
30:     if ShouldAdd(S, open, maxCost) then            ▷ Add the state to the queue
31:         Add(open, S)
32:     return S
```

The AddNeighbors function detailed in Alg. 2.2 adds all the children states from a parent based on the available legal moves —called from Alg. 2.1:24. Given the set of enabled transitions of the parent state (Alg. 2.2:2), and the next trace activity of the parent state

(Alg. 2.2:4), neighbors are created by executing all available legal moves. Specifically, one neighboring state is generated for each move:

- A synchronous move for each enabled transition of the model that shares the label with the next activity of the trace —note that model transitions can have duplicate labels— (Alg. 2.2:6-7).
- An asynchronous movement in the log if the end of the trace is not yet reached (Alg. 2.2:10).
- As many asynchronous movements in the model as transitions are enabled from the current marking of the model (Alg. 2.2:12-13).

The conditions at Alg. 2.2:9 and Alg. 2.2:11 are optimizations. These optimizations reduce the exploration of unnecessary states by skipping certain asynchronous moves in the log and the model when specific conditions are met (Section 2.3.2). For each move, a new neighboring state S must be created and $S.\gamma$ must be updated, the sequence of legal moves (Alg. 2.2:15-22). Each call to AddMove defines the new state that can advance to the next activity on the trace, and/or execute a transition in the model, depending on the kind of move (Alg. 2.2:23-26). The cost and heuristic values are then updated for the new state, just before inserting it in the priority queue, where it will be sorted by the sum of both values (Alg. 2.2:27-31). The condition at Alg. 2.2:30 avoids inserting in the open queue states that are not capable of reaching the optimal alignment, as they match a previously discovered state —equal $S.M$ and $S.i$ values— with a higher or equal $S.c$, or they exceed the $S.c + S.h$ threshold established by the greedy algorithm.

To achieve optimality, the A* algorithm employed in REACH must meet three conditions. First, the cost change when advancing to a neighbor must be strictly positive. To satisfy this requirement, the cost of synchronous and silent moves is always a negligible value greater than 0 ($epsilon$). A standard cost of 1 is applied to all other asynchronous moves. Second, the heuristic must be admissible, meaning that it must return an underestimate of the remaining cost to the closest goal node in terms of cost. In third place, the heuristic must also be consistent —also called monotone— in order to provide optimal results. A consistent heuristic returns an estimate for any node that is lower or equal to the least cost of advancing to a neighbor plus its heuristic estimate, $S_{parent}.h \leq S.c - S_{parent}.c + S.h$, where S is a state successor of S_{parent}. This is because using a consistent heuristic ensures that once a node is explored, it will not be reached again with a lower cost. Note that a consistent heuristic is also admissible. The admissibility and monotonicity of the proposed heuristic will be discussed in Section 2.3.1.

2.3.1 Heuristic

Heuristics substantially affect the performance of the A* algorithm and, as such, are the focus of many state-of-the-art approaches. We propose a new heuristic, called Model Move Required (MMR) that balances the time required to compute the heuristic —which would mean a higher processing time per state—, and the accuracy of the estimates it provides —which enables reducing the number of states that need to be explored to achieve optimality. It works by finding a subset of the required transitions —transitions that must be executed to reach the end of the model— and by comparing their labels to the remaining trace activities.

Alg. 2.3 presents the MMR heuristic, which calculates an optimistic approximation of the cost to complete an alignment from a given state. The heuristic needs to identify a subset of the non-silent transitions that are necessary to execute from the current marking to reach a final state (Alg. 2.3:2). The first step is to figure out the required transitions for the state for which the heuristic will be computed (Alg. 2.3:8). Each token of that state marking is added to the queue of $places$ to visit (Alg. 2.3:12-14). Then, the main loop starts visiting each $place$ of that queue until it is empty (Alg. 2.3:15). Each visited $place$ is removed from the queue, and already visited places are skipped or otherwise marked as visited (Alg. 2.3:16-19). Next, $place\bullet$ is retrieved, i.e., the successor $transitions$ of $place$. If the list only contains one transition, it is registered as a required transition (Alg. 2.3:20-24). All the successor places of the transition are added to the queue of $places$ to visit (Alg. 2.3:25). Table 2.3 presents an example that illustrates the behavior of the required transitions procedure.

When the queue of $places$ to visit is empty, the algorithm collected a subset of all the required transitions. Following the collection of the required transitions, their unique labels are compared with the remaining distinct activities in the trace to calculate the heuristic (Alg. 2.3:4). For each mandatory model activity that does not appear in the remaining trace (Alg. 2.3:5), the cost of the asynchronous move is added to the heuristic. These activities are required to be executed in order to reach the end of the model, so not having them in the remaining trace implies at least another move in the model in order to reach the end. For each of the unmatched trace activities remaining (Alg. 2.3:6), the cost of a synchronous move is added, as another move with a minimum cost of a synchronous move has to be made for each of them in order to complete the alignment.

Algorithm 2.3: Model Move Required heuristic

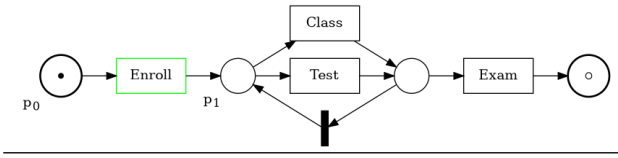
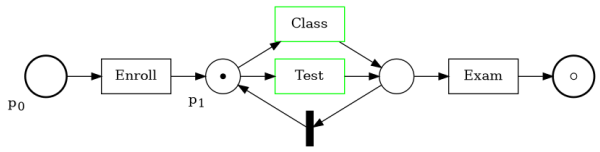
Input: An state S.
Output: The heuristic value.

1: **procedure** HEURISTIC(σ, S)
2: $requiredTrs \leftarrow$ REQUIREDTRANSITIONS($S.M$) ▷ Subset of the required transitions
3: $required \leftarrow \{\lambda(t) \mid t \in requiredTrs\}$ ▷ Set of required unique activities
4: $remaining \leftarrow$ TOSET($\sigma[S.i:]$) ▷ Set of activities of the trace suffix starting at $S.i$
5: $missing \leftarrow required \setminus remaining$ ▷ Set of required activities that were not found
6: $minCostMoves \leftarrow remaining \setminus missing$ ▷ Set of required extra moves
7: **return** $|missing| + |minCostMoves| * epsilon$ ▷ Heuristic
8: **procedure** REQUIREDTRANSITIONS(M)
9: $requiredTrans \leftarrow \emptyset$ ▷ This function returns the set of required transitions
10: $places \leftarrow \emptyset$ ▷ Queue that stores places to visit
11: $placesVisited \leftarrow \emptyset$ ▷ Queue that stores visited places
12: **for all** $place \in$ TOKENPLACES(M) **do** ▷ For each token place in the current marking
13: ADD($places, place$) ▷ Insert the initial place in the queue to visit it later
14: **end for**
15: **while** ¬EMPTY($places$) **do** ▷ While there are more places to visit
16: $place \leftarrow$ POLL($places$) ▷ Extract the next place to explore
17: **if** $place \in placesVisited$ **then**
18: **continue** ▷ Skip the place if already visited
19: ADD($placesVisited, place$) ▷ Mark the place as visited
20: $transitions \leftarrow place\bullet$ ▷ Get the successors of the place
21: **if** $|transitions| \neq 1$ **then**
22: **continue** ▷ Skip the place if it has more than one output arc or no output arcs
23: **if** ¬ISSILENT($transitions[0]$) **then** ▷ If the transition is not silent
24: ADD($requiredTrans, transitions[0]$) ▷ Mark the transition as required
25: ADDALL($places, transitions[0]\bullet$) ▷ Queue each output place for exploration
26: **end while**
27: **return** $requiredTrans$

Example 2.4: Retrieval of required transitions

Table 2.3 shows how the internal function of REQUIREDTRANSITIONS works to provide the mandatory transitions to the heuristic. This figure focuses on the initial state of the running example. For each iteration —row of the table— of the main while loop (Alg. 2.3:15), the place being explored is highlighted on the left column and the state of the defined variables is shown on the right column. The $requiredTrans$, $places$

Table 2.3: Iterations of REQUIREDTRANSITIONS (Alg. 2.3:8) for the initial state of the running example.

Place	State
	$requiredTrans$: $\emptyset$ $places$: <p_0> $placesVisited$: $\emptyset$ $place$: p_0 $transitions$: $[Enroll]$
	$requiredTrans$: $[Enroll]$ $places$: <p_1> $placesVisited$: $[p_0]$ $place$: p_1 $transitions$: $[Class, Test]$

and $placesVisited$ variables show the values before the iteration, while the $place$ and $transitions$ variables show the values retrieved during the iteration. The initial marking only has one token on p_0, which is added to $places$. The $place$ to visit in the first iteration (p_0) is extracted from $places$. This iteration checks that the successor of p_0 is only one transition. As this is true ($|transitions| = 1$), it marks the transition ($Enroll$) as required and adds all output places of the transition ($[p_1]$) to $places$. The second iteration processes p_1 as it is the first element of the $places$ queue. It has two successors, so it does not add any more places to $places$. It does not mark them as required as only one of them has to be executed so neither is mandatory. The $places$ queue is empty, so the algorithm stops. The only required transition found for this simple example is $Enroll$.

In order for this heuristic to be valid, there must be only one reachable final marking of the model: one token in the end place. This condition, known as proper completion, guarantees that all transitions identified through the heuristic procedure will be necessary to reach the end of the model. Hence, this heuristic assumes that the model is a workflow net with proper completion —without the soundness requirement. This is a consistent heuristic that uses the standard cost function: assigns a positive value very close to 0 ($epsilon$) for synchronous moves and a cost of 1 to asynchronous moves. The number of unmatched required activities can only decrease by a maximum of 1 between parent and child states (S_{parent} and S), and it may only decrease by 1 when the cost increases by 1 by performing an asynchronous move

Algorithm 2.4: Partial reachability graph: enabled transitions.

Input: The current marking M. The single partially built reachability graph p for the workflow net is received as input even if the calling pseudocode does not specify it (p is initialized as an empty map). Assumes that EnabledTransitions is called before ExecuteTransition for any marking.

Output: The set of enabled transitions of the marking M.

```
1: procedure EnabledTransitions(M, p)
2:     if M ∈ p then                          ▷ If M had its enabled transitions listed previously
3:         etrs ← {t | (t, m) ∈ p[M]}         ▷ Retrieve the list of enabled transitions from p
4:     else
5:         etrs ← {t | M[t⟩}                  ▷ Compute the enabled transitions from M
6:         p[M] ← {(etr, null) | etr ∈ etrs}  ▷ Record the enabled transitions, mapped to null
7:     return etrs
```

($S.c - S_{parent}.c$), so the consistency condition ($S_{parent}.h \leq S.c - S_{parent}.c + S.h$) is verified.

2.3.2 Optimizing the algorithm

This section explains in detail the remaining performance contributions of this research to the core A*-based alignments algorithm: *(i)* partial reachability graph, and *(ii)* several states reduction optimizations.

Partial reachability graph

All state-of-the-art algorithms need to execute the workflow net to be able to compute optimal alignments. Thus, all algorithms perform operations over the workflow net for creating an initial marking (Alg. 2.1:17), listing all enabled transitions for a marking (Alg. 2.2:2), executing a transition (Alg. 2.2:26) and checking if a marking is final (Alg. 2.1:23). Those operations affect performance, as they are executed several times for each iteration of A*. Our approach introduces a significant difference from the state of the art. We propose the dynamic construction of a partial reachability graph (PRG) as new model markings are reached. This change is aimed at leveraging information from prior model operations to enhance the speed of future iterations. To make the execution of the workflow net faster, a directed graph is created that shows the different explored states of the system. This graph is stored in p, which is initially empty (Alg. 2.4). When explored, each marking is stored as an entry in p

Algorithm 2.5: Partial reachability graph: execute transition.

Input: The current marking M and the transition to execute t. The single partially built reachability graph p for the workflow net is received as input even if the calling pseudocode does not specify it (p is initialized as an empty map). ENABLEDTRANSITIONS assumes that ENABLEDTRANSITIONS is called before EXECUTETRANSITION for any marking.

Output: The new marking M' after executing t on the marking M.

```
1: procedure ExecuteTransition(M, t, p)
2:     if p[M][t] ≠ null then                ▷ If t was previously executed from M
3:         M' ← p[M][t]                      ▷ Retrieve the next marking from the partial reachability graph
4:     else
5:         M' ← M − •t + t•                  ▷ Compute the new marking after executing the t transition
6:         p[M][t] ← M'                      ▷ Record the transition execution in the partial reachability graph
7:     return M'
```

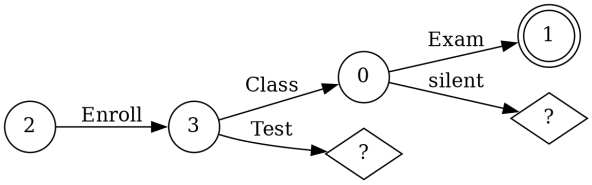


Figure 2.3: Partial Reachability Graph example.

and represents a vertex in the graph (Alg. 2.4:5-6). When each marking M is explored, each enabled transition from M is represented by an entry in the $p[M]$ mapping, which maps to the new marking after executing the transition (Alg. 2.5:5-6), or null if it was never executed (Alg. 2.4:5-6). The enabled transitions serve as arcs connecting states within the graph. This builds at runtime a data structure similar to a reachability graph [97]. Nevertheless, only the explored states are generated. Hence, it avoids building the full reachability graph which would be a computationally intensive task, especially for models with lots of branching and concurrent transitions. Instead, it only stores information about states that are needed in the algorithm's exploration. Once repeated model operations start occurring, e.g. due to loops in the model or to the beginning of the alignment computation over a new trace of the log, the partial reachability graph can quickly access enabled transitions and new markings (Alg. 2.4:2-3 and 2.5:2-3). This information is kept while aligning all the traces of the log to the model, which is the main reason for the speedup.

Algorithm 2.6: LessStatesLog (SRModel).

Input: The trace σ and an state S.
Output: A boolean indicating whether to skip generating log move states from S.

1: **procedure** LessStatesLog(σ, S)
2: **return** |EnabledTransitions$(S.M)|> 0$ and
3: |EnabledTransitions$(S.M) \cap \bigcup\{\lambda_r(act) \mid act \in \sigma[S.i:]\}|= 0$

Example 2.5: Partial Reachability Graph

Figure 2.3 shows an example of the partial reachability graph computed while executing the trace $\langle$*Enroll*, *Class*, *Exam*$\rangle$ on the running example (Figure 1.4). Before executing each transition of the example trace, all enabled transitions of the marking are computed. The nodes of the graph are markings. Known markings are circles and unexplored ones are diamonds. Final markings are represented with a double circle. The node contents show a unique identifier if explored and a question mark otherwise. Each arc between two nodes is labeled with the enabled transition whose execution generates the target marking from the source marking.

The advantages of this PRG become evident as the algorithm progresses and repeated model operations become commonplace. For instance, when loops occur within the model or when aligning a new trace from the log, the PRG swiftly facilitates access to enabled transitions and the corresponding new markings.

State reduction-based optimizations

The primary challenge in computing conformance checking alignments using the A* algorithm is the substantial number of generated states that must be stored and evaluated to ensure optimal results. To mitigate this problem, we propose new optimizations focused on states reduction that will alleviate the state explosion that occurs for complex datasets: *(i)* States Reduction forcing asynchronous Model moves (SRModel), *(ii)* States Reduction forcing asynchronous Log moves (SRLog), and *(iii)* an initial greedy search that finds a cost limit.

The SRModel optimization is only applicable when seeking a single optimal alignment, and it effectively reduces the number of generated neighbors by constraining allowed movements (Alg. 2.2:9, Alg. 2.6). Thus, if the current state has enabled transitions and their activities do not appear in the remaining trace (Alg. 2.6:3), this optimization can be activated: the asynchronous

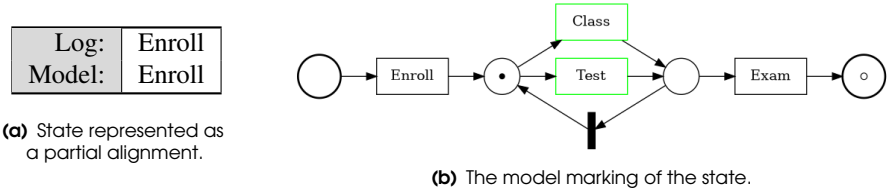
Algorithm 2.7: LESSSTATESMODEL (SRLog).

Input: The trace σ and an state S, for which ALIVEACTIVITIES returns the alive activities from that state (Alg. 2.8).

Output: A boolean indicating whether to skip generating model move states from S.

1: **procedure** LESSSTATESMODEL(σ, S)

2: | **return** $\neg$LESSSTATESLOG(σ, S) and $i < |\sigma|$ and $\sigma[i] \notin$ ALIVEACTIVITIES($S.M$)



(a) State represented as a partial alignment.

(b) The model marking of the state.

Figure 2.4: SRModel optimization example.

move in the log can be avoided, as the optimal alignment must have a model move on one of the enabled transitions. This optimization always provides an optimal alignment because a synchronous or model move is required to reach the end of the model, but no synchronous move is or will ever be available —the remaining trace activities do not match the labels of the currently executable transitions in the model— until a model move is performed. Forcing the model moves in this situation reduces the number of states to explore without affecting the optimality of the algorithm.

Example 2.6: SRModel optimization

Figure 2.4 illustrates an example of applying the SRModel optimization for the trace $\langle Enroll, Exam \rangle$ and the model from Figure 2.1(a). The explored state of Figure 2.4(a) would normally generate three asynchronous moves: one model move for each of the enabled model transitions, and one log move on the *Exam* activity of the trace. However, the SRModel optimization checks whether the enabled model transitions cannot be executed in the remaining trace. As this is true for the given state, it can force a model move by only generating two of the three neighboring states.

Similarly to SRModel, the SRLog optimization reduces the number of states to explore by studying the remaining trace and model. It identifies the alive transitions of the model,

which are those that can be enabled from the current marking through valid transition execution sequences. Alive activities are the unique labels of the alive transitions. SRLog checks if the next activity of the trace —which must have one or more activities left— is not one of the alive activities of the current state. In this case, the algorithm forces an asynchronous movement on the log (Alg. 2.2:11, Alg. 2.7) because that movement has to be made in order to reach a complete alignment. This optimization always provides an optimal alignment because a synchronous or log move is required to reach the end of the trace, but no synchronous move is or will ever be available —the alive activities of the model do not match the next activity in the trace— until a log move is performed. By forcing the algorithm to make it as soon as possible, all states that would otherwise need to be generated later are being avoided. In cases where both SRModel and SRLog are applicable to a particular state, we exclusively employ SRModel. The simultaneous application of both would result in the filtering out of all neighboring states. SRLog is applied to the generated neighbors if SRModel is not available for them. SRLog requires an additional model pre-processing task in which all the alive activities of all states are stored. This pre-processing task is only needed once per model, and it is performed in two phases as is shown in Alg. 2.8.

The initial phase of SRLog involves exploring the model and recording all the unique reachable markings within the workflow net. Each of those markings is associated with a single state, which also includes information about the activities that are directly enabled, and the predecessor states (Alg. 2.8:2). This exploration phase stops when a repeated marking or a marking without enabled transitions is reached.

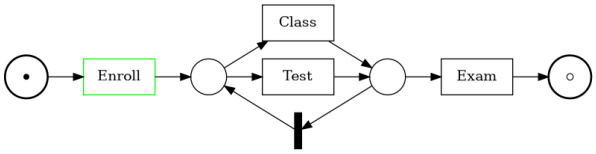
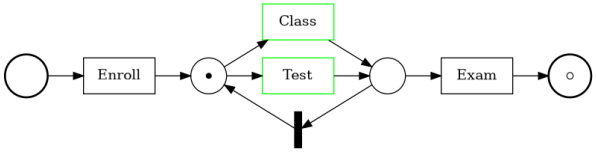
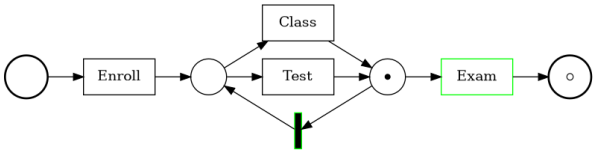
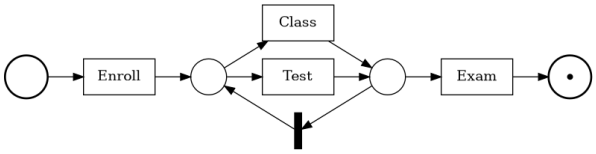
In this phase, a list to store the discovered states that still need to be processed is used (Alg. 2.8:5). Each iteration explores a new discovered state, and the loop stops when there are no more states in the queue (Alg. 2.8:9-10). The first action for each discovered state is to check if its marking was previously explored, registering it otherwise. This is performed by PutIfAbs, which also returns the previously stored state for that marking if it exists (Alg. 2.8:11). This previously stored state will later be used to merge already discovered predecessor states and enabled activities into the single state associated with that shared marking (Alg. 2.8:25-27). To further explore new states, it is necessary to iterate over each enabled transition of the current state which might lead to a new child state (Alg. 2.8:13-15). It is checked if the current child marking is new by comparing to previously explored markings. In either case, the label of the transition is registered as an alive activity of the parent state and the predecessors of the child state are updated (Alg. 2.8:16-22).In the event that the child state is indeed new, it is added to

Algorithm 2.8: Initialization for LESSSTATESMODEL (SRLog).

Input: A process model PN.
Output: The map from each reachable marking to the set of alive activities in the model.

procedure LESSSTATESMODELINIT(PN)
  $S \leftarrow \{ M : \text{INITIALMARKING}(PN),$
    $alive : \emptyset,$ ▷ Alive activities from S
    $pred : \emptyset \}$ ▷ Predecessor states
  $states \leftarrow list()$ ▷ Unexplored state list
  $explored \leftarrow map()$ ▷ Marking to state mapping
  $finalState \leftarrow null$
  ADD($states, S$)
  while $|states| > 0$ **do** ▷ Phase 1: discovery
    $S \leftarrow \text{POLLLAST}(states)$ ▷ Retrieve and remove the last element of the $states$ list.
    $S_{prev} \leftarrow \text{PUTIFABS}(explored, S.M, S)$ ▷ Save the state only if absent
    $etrs \leftarrow \text{ENABLEDTRANSITIONS}(S.M)$ ▷ Alg. 2.4
    for all $etr \in etrs$ **do** ▷ Explore all enabled transitions
      $S_c \leftarrow \text{NEWSTATE}(S)$
      $S.M \leftarrow \text{EXECUTETRANSITION}(S_{parent}.M, tr)$ ▷ Alg. 2.5
      $S_{back} \leftarrow \text{GET}(explored, S_c.M)$ ▷ Find out if this was previously explored
      if $S_{back} \neq null$ **then** ▷ Handle discovered loops to previous states
        if $etr \in T_{ns}$ **then** $S.alive \leftarrow S.alive \cup \{\lambda(etr)\}$
        $S_{back}.pred \leftarrow S_{back}.pred \cup \{S\}$
      else ▷ Handle new states by updating $alive$ and $pred$, and adding them to $states$
        if $etr \in T_{ns}$ **then** $S.alive \leftarrow S.alive \cup \{\lambda(etr)\}$
        $S_c.pred \leftarrow S_c.pred \cup \{S\}$
        ADDLAST($states, S_c$) ▷ Queue for exploration by adding them to $states$
    end for
    if $S_{prev} \neq null$ **then** ▷ Merge collected data if this is a previously explored state
      $S_{prev}.alive \leftarrow S_{prev}.alive \cup S.alive$
      $S_{prev}.pred \leftarrow S_{prev}.pred \cup S.pred$
    else if ISFINAL($S.M$) **then** ▷ Remember the final state
      $finalState \leftarrow S$
  end while
  $explored \leftarrow map()$ ▷ Marking to state mapping
  ADD($states, finalState$)
  while $|states| > 0$ **do** ▷ Phase 2: alive activities
    $S \leftarrow \text{POLLLAST}(states)$
    $S_{prev} \leftarrow \text{PUTIFABS}(explored, S.M, S)$ ▷ Save the state only if absent
    for all $S_c \in S.pred$ **do** ▷ Explore predecessor states
      $prevAlive \leftarrow S_c.alive$ ▷ Remember previously alive activities
      $S_c.alive \leftarrow S_c.alive \cup S.alive$ ▷ Inherit previously alive activities
      if $S_{prev} = null$ **or** $S_c.alive \neq prevAlive$ **then** ▷ Check stop condition
        ADDLAST($states, S_c$) ▷ Mark for exploration
    end for
  end while
  return $explored$

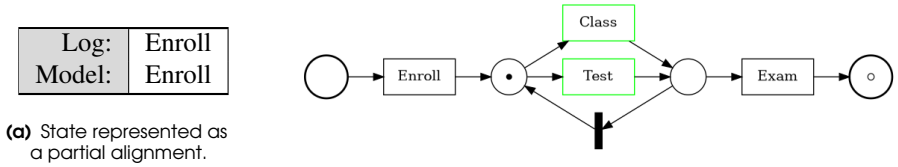
Table 2.4: Example of execution steps of Alg. 2.8 (initialization for SRLog). The S and A subscripts help to distinguish between state and activity IDs respectively.

Marking	State
	PHASE 1: State ID: 0_S Predecessors: $\emptyset$ Enabled: $\{0_A\}$ PHASE 2: Alive: $\{0_A, 1_A, 2_A, 3_A, 4_A\}$
	PHASE 1: State ID: 1_S Predecessors: $\{0_S, 2_S\}$ Enabled: $\{1_A, 2_A\}$ PHASE 2: Alive: $\{1_A, 2_A, 3_A, 4_A\}$
	PHASE 1: State ID: 2_S Predecessors: $\{1_S\}$ Enabled: $\{3_A, 4_A\}$ PHASE 2: Alive: $\{1_A, 2_A, 3_A, 4_A\}$
	PHASE 1: State ID: 3_S Predecessors: $\{2_S\}$ Enabled: $\emptyset$ PHASE 2: Alive: $\emptyset$

where activity IDs are {*Enroll*: 0_A, *Class*: 1_A, *Test*: 2_A, τ: 3_A, *Exam*: 4_A}

the state queue to be explored later (Alg. 2.8:23). The final state is also recorded, which will be the starting point for phase 2 of this algorithm (Alg. 2.8:29).

During the second phase of SRLog, the algorithm calculates all alive activities associated with each marking. These are activities that might be executed at any point in the future, even if they are not enabled at this point and require executing other activities first. This is done by recursively inheriting all the alive activities from states to their predecessors, starting from the recorded final state. The *states* list is initialized to the final state (Alg. 2.8:32). The main loop starts, taking the last state from the *states* list as long as it is not empty (Alg. 2.8:33). For each explored state, S_{prev} is not null if and only if $S.M$ was previously explored, and $S.M$ is marked as explored (Alg. 2.8:35). Then, all predecessor states of S inherit the alive tasks from S (Alg. 2.8:38). The predecessors are appended to the end of the *states* list if they were not previously explored or if new enabled tasks were found (Alg. 2.8:40).



(a) State represented as a partial alignment.

(b) The model marking of the state.

Figure 2.5: SRLog optimization example

Example 2.7: SRLog optimization

Table 2.4 shows step-by-step the execution of the initialization of SRLog over the running example. During the first phase of this initialization algorithm, the following information for each state is obtained: its ID, the IDs of its predecessor states, and the directly enabled activities. The second phase completes the full set of alive activities by inheriting enabled activities from successors. This results in the realization that all activities are alive at the initial marking, all activities except for *Enroll* are alive for the intermediate markings, and no activities are alive when the final marking is reached.

In addition, Figure 2.5 shows the execution of the SRLog algorithm for the running example from Figure 2.1(a) and the trace $\langle Enroll, Enroll, Class, Exam \rangle$. The alive transitions computed using Alg. 2.8 (Table 2.4) for the current state are *Class*, *Test*, the silent one, and *Exam*. The explored state of (a) would normally generate three moves: one model move for each of the enabled model transitions, and one log move on the *Enroll* event of the trace. However, the SRLog optimization checks whether the *Enroll* event is not alive in the model. As this is true for the given state, it can force a log move by only generating one of the three neighboring states.

Note that both SRModel and SRLog have no relation to the log-move-first and model-move-first optimizations defined in [1]. The optimizations we introduce involve an analysis of the remaining trace and/or model, identifying mandatory moves and actively enforcing them. This stands in contrast to [1] in which only the last alignment move is examined to dictate the sequence of log and model moves. Both SRModel and SRLog can be enabled at the same time during the execution of the algorithm for improved performance, whereas only one of log-move-first or model-move-first can be applied on each execution. Note that neither

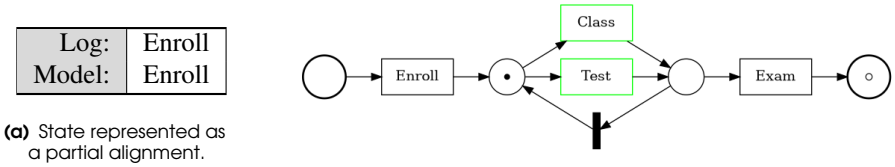
Algorithm 2.9: Greedy alignments algorithm.

Input: The process model PN and a trace σ.
Output: The final state of the greedy alignment, or null if no greedy alignment is found.

```
1:  procedure AlignGreedy(PN, σ)
2:      S ← InitialState(PN, σ)
3:      visited ← ∅
4:      alignment ← null
5:      while alignment = null and S ≠ null do                    ▷ Explore states
6:          Add(visited, (S.M, S.i))          ▷ Mark their marking and trace index as visited
7:          if ¬IsFinal(S) then
8:              open ← ∅   ▷ Reset the open queue on each iteration, disabling backtracking
9:              AddNeighbors(S, open)              ▷ Alg. 2.2: consider all neighbors as usual
10:             while S ≠ null do
11:                 S ← Poll(open)                                ▷ Extract them by priority
12:                 if ¬Contains(visited, (S.M, S.i)) then ▷ Ignore already visited neighbors
13:                     break
14:             end while
15:         else
16:             alignment ← S                                     ▷ Use it as the result if final
17:     end while
18:     return alignment
```

log-move-first nor model-move-first are compatible with the SRModel or SRLog optimizations.

Before the primary alignments algorithm, the greedy search optimization (Alg. 2.9) is employed to quickly establish an upper cost bound for the optimal alignment. The algorithm starts from the same initial state as the REACH algorithm (Alg. 2.9:2-5). It records a tuple of the visited marking and trace progress to quickly advance avoiding infinite loops (Alg. 2.9:6). It generates the neighbors of the initial state also following the REACH algorithm (Alg. 2.2), and adds them to the newly created $open$ queue (Alg. 2.9:8-9). Considering only the neighbors generated on that iteration, the one of minimum $S.c + S.h$ is chosen (Alg. 2.9:11), i.e., the greedy algorithm performs a Best First Search guided by the heuristic. If it was already visited (Alg. 2.9:12), it goes back to Alg. 2.9:10 to process the next best state. If the chosen state was not visited before, the algorithm moves to that state and continues the exploration from there. It does so until a complete alignment is achieved or until there are no more unvisited states left to explore (Alg. 2.9:5). The cost of this suboptimal greedy alignment is used as the upper bound of $S.c + S.h$ for the optimal alignment retrieved by the REACH algorithm, filtering states that will not lead to the optimal solution.



(a) State represented as a partial alignment.

(b) The model marking of the state.

Figure 2.6: Greedy alignment optimization example.

Example 2.8: Greedy optimization

Figure 2.6 shows an example of how this filtering is made for the running example from Figure 2.1(a) and the trace $\langle Enroll, Class, Exam \rangle$. The greedy alignment computed using Alg. 2.9 is made of three synchronous moves. The explored state of (a) would normally generate four moves: one synchronous move, one model move for each of the enabled model transitions, and one log move on the *Class* event of the trace. However, the greedy alignments optimization checks whether the cost of each generated neighbor is greater than the cost of the greedy alignment. As this is true for all of the asynchronous moves of the given state, it can force a synchronous move by only generating one of the four neighboring states.

2.4 Evaluation

We have extensively evaluated our approach and compared it against other state-of-the-art algorithms. We provide REACH as a web service, as a binary executable and as the original source code[1] to allow the replication of results. In this section, we describe the datasets and the experiments, discussing the results.

2.4.1 Setup

To ensure consistency, we developed and tested REACH in Java 8, aligning it with the Java-based implementations of the state-of-the-art algorithms. All the algorithms have been executed in an Oracle Java 8 Virtual Machine in a computer equipped with an Intel Core i5-9600K, 32

[1] `https://tec.citius.usc.es/reach`

Table 2.5: Information for all event logs in the evaluation dataset. The columns are: number of activities (A), number of traces (T), total number of events (E), minimum number of events per trace (N), and maximum number of events per trace (X). Note that the S subindex indicates that redundant traces are ignored.

Name	A	T	E	T_S	E_S	N	X
BPIC12	24	13,087	262,200	4,366	182,467	3	175
BPIC13CP	4	1,487	6,660	183	1,810	1	35
BPIC13INC	4	7,554	65,533	1,511	29,010	1	123
BPIC14F	9	41,353	369,485	14,948	232,067	3	167
BPIC15F1	70	902	21,656	295	10,367	5	50
BPIC15F2	82	681	24,678	420	17,820	4	63
BPIC15F3	62	1,369	43,786	826	28,553	4	54
BPIC15F4	65	860	29,403	451	16,502	5	54
BPIC15F5	74	975	30,030	446	18,789	4	61
BPIC17F	18	21,861	714,198	8,767	333,965	11	113
BPIC18	41	43,809	2,514,266	28,457	1,819,830	24	2,973
BPIC19_1	11	1,044	5,898	148	1,619	2	21
BPIC19_2	38	15,182	319,233	4,228	255,982	1	990
BPIC19_3	39	221,010	1,234,708	7,832	82,611	1	179
BPIC19_4	15	14,498	36,084	281	1,614	1	17
RTFMP	11	150,370	561,470	231	1,891	2	20
SEPSIS	16	1,050	15,214	846	13,775	3	185
sepsis [98]	16	1,050	15,214	846	13,775	3	185
Fitting logs [99]	42	4,000	83,402	2,935	76,482	5	102
Noisy logs [99]	42	16,000	322,670	12,051	298,281	2	102
Fitting logs [100]	317	1,200	49,792	1,126	48,573	14	59
Noisy logs [100]	363	5,300	638,555	5,149	633,965	15	245
Fitting logs [82]	110	34,000	1,062,208	22,649	906,695	12	167
Noisy logs [82]	68	32,000	1,035,889	22,747	910,515	12	147
bpi12 [101]	24	13,087	262,200	4,366	182,467	3	175
road_fines [102]	11	150,370	561,470	231	1,891	2	20

GB RAM, 1024 GB SSD, and Ethernet 1Gb BaseT. For the execution of each algorithm, a maximum of 24 GB RAM was reserved.

2.4.2 Logs and process models

The algorithm takes two inputs: a log and a model, referred to as a log-model pair for the sake of simplicity. We used logs and models from the following previous works:

- [95] includes 17 logs extracted from different years of the Business Process Intelligence Challenge (BPIC) and from 4TU.ResearchData[2]. They are taken from different domains such as healthcare, government, finance and IT service management, with different levels of complexity to create a fair test environment for conformance checking algorithms.

  The process models were discovered with the Inductive Miner infrequent algorithm [103]. Discovering models that perfectly fit the log would be too easy to solve as the optimal alignment would only require synchronous moves. To increase the difficulty and test the performance limits of the state of the art, we discover models with various fitness levels. These models differ in the extent to which they describe the log behavior, with lower percentages indicating lower fitness levels. Achieving optimal alignments generally becomes more challenging as the fitness decreases. Thus, for each log, 10 models will be discovered, ranging from 10% to 100% of behavior[3]. For the log BPIC18, the discovery algorithm could not obtain a model for percentages over 30%, so only 3 models were discovered. The total of log-model pairs extracted from this source is 163.

- [41] includes logs and models, so no model discovery algorithm was required. The main advantage of including these log-model pairs is that they include models discovered with different algorithms and even large artificial models. From this source, we expanded our dataset with an additional 64 log-model pairs.

Adding the log-model pairs obtained from [95] and [41] results in a total of 227 log-model pairs. The complete set of logs used in our evaluation, along with relevant statistics showcasing their diversity, is presented in Table 2.5.

2.4.3 Impact of the optimizations in REACH

In this section, we analyze the effect of the proposed optimizations on the performance of the A* algorithm by selectively enabling and disabling the optimizations. Note that we always

[2] https://data.4tu.nl/

[3] In this chapter, when we mention the percentage of considered behavior, we are referring to the complement of the threshold parameter of the Inductive Miner infrequent algorithm.

check the optimality of the algorithm for each experiment by comparing the returned alignment costs against other versions of the algorithm or against the state of the art.

We tested the proposed optimizations of Section 2.3.2. For each combination of optimizations, we verified that the algorithm produced alignments of consistent cost and evaluated the enhancements in terms of: *(i)* the number of states generated before reaching an optimal solution; and *(ii)* the processing time. If the execution for a single log-model pair takes longer than 5 minutes for any combination of optimizations, we do not consider it to keep the test times reasonable and the statistics fair.

Table 2.6 shows performance statistics for the complete dataset. Performance has been evaluated in terms of the average execution time in milliseconds (time) and the average number of states discovered (states). To better display the performance increase, both measurements are divided into simple and complex log-model pairs as indicated by the subscript. Simple log-model pairs are those that take less than 10 seconds to solve without optimizations. In addition, both metrics are taken as absolute values —(a) and (b)— and as percentages of improvement with respect to the execution with no optimizations applied —(c)).

The partial reachability graph (PRG) optimization is aimed at reducing the computation time required for the workflow net execution. Hence, it does not change the total number of states that need to be processed when enabled. It improves execution times by reducing the number of operations when interacting with the process model. The PRG optimization demonstrates its maximum effectiveness when no other optimizations are enabled, particularly when aligning complex log-model pairs, resulting in an average execution time reduction from 29.0 to 27.8 seconds. This improvement is reduced when other optimizations are enabled but, even then, the PRG optimization does not have a negative effect on the average execution time. It is affected by other optimizations as they lower the number of model operations by reducing the number of discovered states.

The States Reduction forcing asynchronous Model moves (SRModel) optimization achieves a reduction in the number of states discovered at the cost of increasing the processing time per state. For simple log-model pairs, which are those solved in under 10 seconds without any optimizations, SRModel reduces the average number of states by 8.6%. For complex problems, the reduction is even more substantial, amounting to 16.4% fewer states. For simple log-model pairs, the reduction of states is on par with the increased time per state, leading to approximately the same mean total time (1.7% decrease in execution time). Nevertheless, it should be noted that this states reduction has a much greater effect on the complex log-model pairs, resulting in

Table 2.6: Impact of the states reduction optimizations and the partial reachability graph optimization (PRG) on the performance of the REACH algorithm for the complete dataset.

(a) Absolute metrics with different optimizations enabled.

PRG				
SRModel		✓		✓
SRLog			✓	✓
$\text{states}_{\text{simple}}$	$1.73 \cdot 10^6$	$1.58 \cdot 10^6$	$1.47 \cdot 10^6$	$1.31 \cdot 10^6$
$\text{states}_{\text{complex}}$	$2.96 \cdot 10^7$	$2.47 \cdot 10^7$	$2.64 \cdot 10^7$	$2.00 \cdot 10^7$
$\text{time}_{\text{simple}}$	$3.02 \cdot 10^3$	$2.96 \cdot 10^3$	$2.65 \cdot 10^3$	$2.66 \cdot 10^3$
$\text{time}_{\text{complex}}$	$2.90 \cdot 10^4$	$2.51 \cdot 10^4$	$1.97 \cdot 10^4$	$1.73 \cdot 10^4$

(b) Absolute metrics with different optimizations enabled.

PRG	✓	✓	✓	✓
SRModel		✓		✓
SRLog			✓	✓
$\text{states}_{\text{simple}}$	$1.73 \cdot 10^6$	$1.58 \cdot 10^6$	$1.47 \cdot 10^6$	$1.31 \cdot 10^6$
$\text{states}_{\text{complex}}$	$2.96 \cdot 10^7$	$2.47 \cdot 10^7$	$2.64 \cdot 10^7$	$2.00 \cdot 10^7$
$\text{time}_{\text{simple}}$	$3.02 \cdot 10^3$	$2.97 \cdot 10^3$	$2.66 \cdot 10^3$	$2.66 \cdot 10^3$
$\text{time}_{\text{complex}}$	$2.78 \cdot 10^4$	$2.46 \cdot 10^4$	$2.00 \cdot 10^4$	$1.74 \cdot 10^4$

(c) Relative improvement (%).

PRG	✓	✓	✓
SRModel	✓		✓
SRLog		✓	✓
$\text{states}_{\text{simple}}$	8.6	15.1	24.0
$\text{states}_{\text{complex}}$	16.4	10.5	32.4
$\text{time}_{\text{simple}}$	1.7	12.2	11.9
$\text{time}_{\text{complex}}$	15.3	31.2	40.1

a reduction of the average time of 15.3%. Note that these percentages are measured with the PRG optimization enabled, as REACH will also enable all optimizations. The improvements in states discovered and execution time remain consistent with the PRG optimization disabled. This means that the SRModel optimization helps REACH to solve harder problems, without losing average performance on simple models. The SRModel optimization has a greater impact

Table 2.7: Slowest initialization times in milliseconds for the SRLog optimization.

Log	Model	Time (ms)
BPIC19_2	100%	89,502
BPIC15F5	60%	18,951
BPIC15F5	70%	18,746
BPIC15F5	50%	18,334
BPIC15F5	100%	13,441
BPIC15F5	80%	12,675
BPIC15F5	90%	12,394
BPIC15F5	40%	11,389
BPIC15F5	30%	10,845
BPIC15F5	20%	8,641
SEPSIS	90%	166

for bigger models, especially when the alignment between the model and the trace is very poor, which happens when the fitness is lower, i.e., for models that do not support much behavior of the log.

The States Reduction forcing asynchronous Log moves (SRLog) optimization reduces the number of discovered states by 15.1% for simple models and 10.5% for complex models. It outperforms SRModel in state reduction for simple models, while SRModel achieves better state reduction for complex models. Nevertheless, SRLog stands out for its performance improvements for more complex models, reaching an average reduction of 31.2% of the execution time. SRLog is more effective in models with many branches since forcing an asynchronous move in the log will remove all the states generated by each enabled activity on the model. As SRModel, this optimization has a great impact on models with low fitness since the probability that the activities of the remaining trace do not appear in the model is much higher. Its performance improvement is also evident in simple models, with an average execution time reduction of 12.2%. This is due to the fact that a significant portion of the computational cost of SRLog can be executed during an initialization phase, thereby reducing the time spent on each state.

The SRLog optimization requires an initialization phase whose execution time is generally very low for most of the log-model pairs. This initialization has been taken into account in the times reported on Table 2.6. To further investigate this initialization time, Table 2.7 shows in descending order the 11 highest execution times for the initialization of the SRLog

optimization, out of all 163 log-model pairs from [95]. The model column indicates what percentage of behavior the model supports. For all other log-model pairs, the execution times of this initialization are less than 100 milliseconds. Therefore, the SRLog optimization incurs negligible penalties in terms of execution time. Moreover, it is worth mentioning that in almost all cases in which the initialization takes more than 200 milliseconds —8 out of 10—, the algorithm would still take more than 5 minutes to finish if the optimization was disabled, so SRLog is improving the algorithm's performance. In the other 2 log-model pairs —BPIC15F5 with models with 100% and 90% behavior supported—, the initialization takes less than 15 seconds while the algorithm takes around 28 seconds.

SRModel and SRLog can be combined to complement each other and avoid redundant tasks. Enabling both optimizations results in an even better reduction in the number of discovered states —24.0% for simple models and 32.4% for complex models— and total execution time —11.9% for simple log-model pairs and 40.1% for complex ones.

In conclusion, the proposed optimizations for reducing the number of discovered states improve very significantly the efficiency of REACH over the test dataset, being the cause of its high performance when compared to the state-of-the-art proposals.

2.4.4 Comparison with the state of the art

We compare REACH with several publicly available algorithms of the state of the art. We have included techniques that obtain the optimal alignments —ProMNoILP [32] (ProM, PNetReplayer v6.11.191), ProMILP [39] (ProM, PNetReplayer v6.11.191), ProMLP [1] (ProM, PNetReplayer v6.11.191), eMEQ [41] (incremental version, ProM, Alignment v6.10.122), RecomposingReplay [67] (ProM, DecomposedReplayer v6.9.97), PartialReplayer [38] (ProM, PartialOrderReplayer v6.9.177), AutoConf [68] (AutomataConformance v1.2 [95])—, and techniques that do not necessarily return the optimal alignments —AutoSComp [66] (AutomataConformance v1.2), AutoTRSComp [95] (AutomataConformance v1.2) and AutoHybrid [95] (AutomataConformance v1.2).

In the experiments, each algorithm is provided with a log in XES format, and one of its discovered models, in PNML format, and it returns one alignment per trace. All algorithms are configured in multi-thread mode to get the speedup of parallel processing. Furthermore, for each log-model pair, we measure the execution time from the moment the algorithm receives the inputs until it obtains the alignment for each trace of the log. Hence, we also account for the

time needed for parsing the inputs (log and model) and writing the alignments. This is because some algorithms may involve pre/post-processing steps that should not be excluded from the total execution time as it would be unfair to other algorithms. Finally, once the alignments are obtained, for each trace we check whether its alignment is valid and, if applicable, whether its cost is the same as the alignments for that trace returned by the other optimal algorithms. Note that the alignments returned by non-optimal algorithms are considered a solution, regardless of whether their cost for each trace is optimal or not. Although performing the comparison in this way is unfair for optimal algorithms like REACH, the aim is to show whether they obtain better results even with this disadvantage.

For practical reasons, we set a time limit for the computation times of the log-model pairs. We have considered two time limits —10 and 300 seconds—, and for each of them we analyze the performance using several visualizations. The first one is a graph that shows the number of log-model pairs that have been successfully computed —the algorithm returns the best alignment it has found— over time. Note that if an algorithm is not able to compute a log-model pair, e.g., it gets stuck processing a trace or cannot align some structures and throws an error, it is considered as if the time limit was reached. The second visualization is a ranking that compares the time taken by each of the algorithms to align each log-model pair. Furthermore, we also provide tables that show how the fitness and precision of input log-model pairs can affect the number of problems solved by each algorithm, highlighting the algorithm that solves the highest number of pairs within the given fitness or precision range.

Results with a time limit of 10 seconds

Figure 2.7 shows the results of the algorithms for a time limit of 10 seconds. It can be seen that all the algorithms compute the simplest alignments in less than 1 second. However, after that point, the curve representing REACH sharply rises above those of the other algorithms. This indicates that REACH is able to process a significantly larger number of log-model pairs in less time. In just 1.5 seconds, REACH successfully processes 117 pairs. Meanwhile, the second-best algorithm —eMEQ—, which is also optimal, computes 85 pairs, closely followed by the optimal algorithm AutoConf and the non-optimal algorithm AutoSComp, both of which compute 84 pairs in that time. This faster behavior of REACH is even greater for log-model pairs that require more computation time, showing the scalability of our approach. Thus, in 9 seconds REACH is able to successfully complete 204 pairs, while the second-best algorithm —eMEQ— computes 152. Upon reaching the 10-second time limit, REACH has processed 207

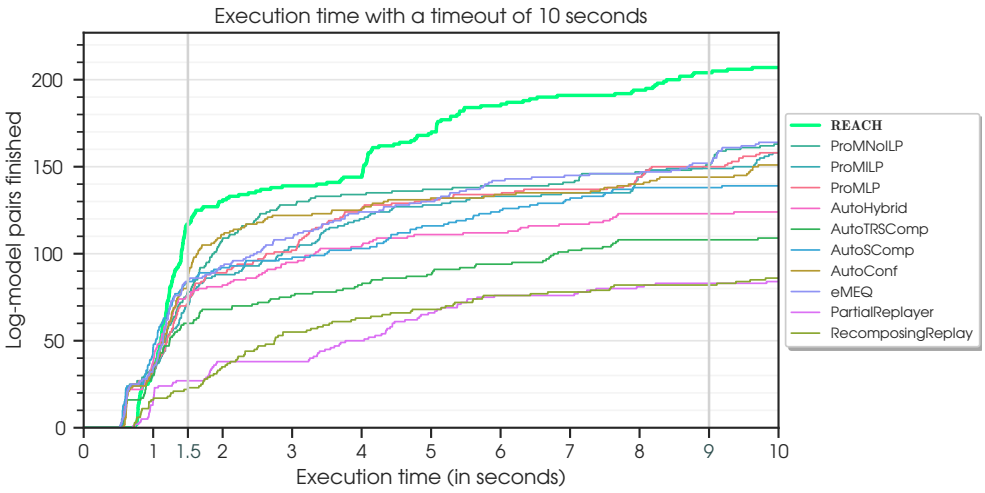


Figure 2.7: Solved problems of each algorithm with a time limit of 10 seconds.

out of 227 log-model pairs, whereas the second-best algorithm, eMEQ, is only able to complete 164 pairs. Note that the next best optimal algorithm —ProMNoILP— achieves a solution for 163 log-model pairs. Therefore, with a limit of 10 seconds, REACH has a performance that is 26% better than the best state-of-the-art algorithm.

Table 2.8 displays the solved log-model pairs per algorithm in a timeout of 10 seconds, categorizing problems by fitness and precision levels. It should be noted that the algorithm used to compute precision is Alignment Based Precision Checking with one optimal alignment [104], and that this algorithm is not able to compute some log-model pairs due to the high execution time and memory requirements —this is totally independent of the alignment algorithm. This analysis allows us to examine how fitness and precision can affect the performance of each alignment algorithm. Regarding fitness, REACH consistently outperforms the state of the art. There is no clear effect on the number of solved problems depending on the fitness value. However, no algorithm can solve log-model pairs fitnesses lower than 0.25 in 10 seconds or less. This outcome is expected because these represent the most challenging problems: a lower fitness means a higher cost of the optimal alignment, and alignments of higher cost —with more misalignments to repair— are much more difficult to compute. In terms of precision, REACH also solves more problems than the state of the art, with the exception of precision values lower than 0.5, where there is only one example solved by all algorithms. Similarly to fitness,

Table 2.8: Number of solved problems of the tested algorithms with a time limit of 10 seconds, splitting log-model pairs by fitness and precision.

Algorithm	Fitness				Precision		
	[0.0, 0.25)	[0.25, 0.5)	[0.5, 0.75)	[0.75, 1.0]	[0.0, 0.5)	[0.5, 0.75)	[0.75, 1.0]
REACH	**0**	**22**	**64**	**121**	**1**	**32**	**62**
ProMNoILP	**0**	16	45	102	**1**	21	55
ProMILP	**0**	15	44	99	**1**	26	52
ProMLP	**0**	16	42	100	**1**	24	53
AutoHybrid	**0**	4	35	85	**1**	16	40
AutoTRSComp	**0**	2	31	76	**1**	16	31
AutoSComp	**0**	15	42	82	**1**	19	49
AutoConf	**0**	9	42	100	**1**	28	53
eMEQ	**0**	16	43	105	**1**	27	60
PartialReplayer	**0**	14	19	51	**1**	10	36
RecomposingReplay	**0**	15	9	62	**1**	16	35
Number of problems	3	28	66	130	1	33	62

the precision level of input log-model pairs does not affect on the number of problems solved by REACH.

We have also compared the execution times that each algorithm needs to solve each of the 227 log-model pairs. To establish whether there are statistically significant differences between REACH and the other tested algorithms, we performed a non-parametric test using the execution times. First, a Friedman's Aligned Ranks test with a significance level of 0.05 has been applied. The ranking is calculated by ordering the execution times of all the algorithms for each log-model pair —the best algorithm obtains a rank of 1—, and then averaging the ranking of each algorithm in all the log-model pairs. The results of this test are summarized in Table 2.9a. The table clearly shows that REACH achieves the top ranking. In the second position, we find a non-optimal algorithm, AutoSComp, followed by the second-best optimal algorithm, PromNoILP.

As the *p-value* of the Friedman test is lower than 10^{-5}, there are significant differences in performances among the algorithms. Thus, we applied Holm's post hoc test to perform a pairwise comparison between REACH and the tested algorithms. The results of this test confirm that there are statistically significant differences between REACH and the other algorithms, with *p-values* consistently lower than 10^{-5}. Therefore, we can conclude that REACH is, on average,

Table 2.9: Performance ranking of the tested algorithms.

(a) With a time limit of 10 seconds.

	rank
REACH	3.405
ProMNoILP	5.084
AutoSComp	5.132
AutoConf	5.460
eMEQ	5.518
AutoHybrid	5.597
ProMLP	5.888
ProMILP	6.081
AutoTRSComp	6.319
RecomposingReplay	8.566
PartialReplayer	8.949

(b) With a time limit of 300 seconds.

	rank
REACH	3.242
ProMNoILP	4.641
eMEQ	4.998
ProMLP	5.390
AutoSComp	5.399
AutoConf	5.678
ProMILP	5.797
AutoHybrid	5.945
AutoTRSComp	6.888
RecomposingReplay	8.980
PartialReplayer	9.042

the fastest algorithm in our experiments over 227 diverse log-model pairs.

Results with a time limit of 300 seconds

Figure 2.8 depicts the results of the algorithms for a time limit of 300 seconds. REACH computes 216 log-model pairs in less than 45 seconds, reaching the time limit in only 11 of them. Conversely, other fast algorithms require more time to compute alignments, delivering results progressively and nearing the 5-minute time limit —and cannot return optimal alignments for some log-model pairs for which our approach is successful. Concretely, after 45 seconds of execution, the second-best optimal algorithms —eMEQ and ProMNoILP— align 195 log-model pairs, and the next best optimal algorithm —ProMILP— solves 194 pairs, while REACH aligns 9% more log-model pairs. When the time limit of 5 minutes is reached, the fastest algorithms after REACH are eMEQ, ProMLP and ProMNoILP, with 214, 212, and 210 solved log-model pairs respectively —in the best case, they solve 2 log-model pairs less than REACH solves in less than 45 seconds.

For the 227 tested log-model pairs, REACH is the fastest optimal algorithm in 125 pairs. In the remaining pairs, the computation time of REACH is, at most, 3 seconds slower than the fastest algorithm for each pair, except for 3 log-model pairs from Noisy logs [100], for which only eMEQ is capable of finishing within the given timeout. These models are synthetic with

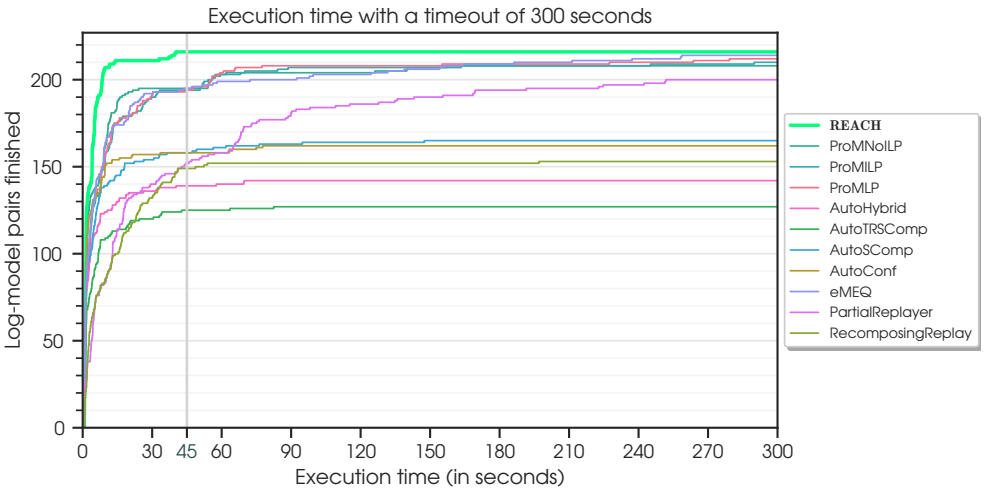


Figure 2.8: Solved problems of each algorithm with a time limit of 300 seconds.

extensive parallelism, leading to an excessive number of states that most algorithms cannot efficiently explore. As stated in [41], the logs for those models were built with vast amounts of swapped activities towards the end of the traces, which is a known weakness of A*-based methods. The ILP-based heuristic proposed in [41] is very effective in detecting swapped activities. However, this method spends more time computing the heuristic on each state, which makes it slower on average in the complete test dataset.

Our algorithm is the only one capable of computing alignments for the most complex models of the BPIC15 log. The primary contributor to the success of these log-model pairs is the SRLog optimization, which can prevent timeouts on its own. These examples have a large number of transitions (around 130), of which a considerable amount are silent transitions (around 60), leading to the generation of a large number of model moves needed from each state explored by the A* algorithm. The proposed optimization benefits from this situation, as it reduces the number of states when possible by forcing an asynchronous move in the log and avoiding all those unnecessary model moves.

Table 2.10 displays the solved log-model pairs per algorithm within a timeout of 300 seconds, categorizing problems by fitness and precision levels. Focusing on fitness, eMEQ and ProMLP solve one more problem than REACHin ranges 0.75-1.00 and 0.25-0.50, respectively, but overall, REACH consistently matches the performance of the best state-of-the-art algorithms.

Table 2.10: Number of solved problems of the tested algorithms with a time limit of 300 seconds, splitting log-model pairs by fitness and precision.

Algorithm	Fitness				Precision		
	[0.0, 0.25)	[0.25, 0.5)	[0.5, 0.75)	[0.75, 1.0]	[0.0, 0.5)	[0.5, 0.75)	[0.75, 1.0]
REACH	**3**	22	**65**	126	**1**	**33**	**62**
ProMNoILP	**3**	22	64	119	**1**	30	60
ProMILP	2	22	64	122	**1**	**33**	61
ProMLP	**3**	**23**	63	123	**1**	32	61
AutoHybrid	0	4	37	101	**1**	25	43
AutoTRSComp	0	2	31	94	**1**	22	38
AutoSComp	0	19	45	101	**1**	27	55
AutoConf	**3**	9	45	105	**1**	29	55
eMEQ	**3**	22	62	**127**	**1**	**33**	**62**
PartialReplayer	0	21	60	119	**1**	32	61
RecomposingReplay	0	17	39	97	**1**	29	50
Number of problems	3	28	66	130	1	33	62

We have observed no correlation between the fitness range of the input problem and the number of problems solved by REACH. In terms of precision, REACH is matched by eMEQ, while algorithms like ProMILP, PromLP or PartialReplayer exhibit very similar performance solving one or two less log-model pairs than REACH. Again, we have observed no correlation between the precision range of the input problem and the number of solved problems of REACH. As shown in Figure 2.8 even though REACH solves almost the same number of problems as some algorithms, it does so much faster than the state of the art.

In the rankings (Table 2.9b), REACH holds a position with a score of 3.242, surpassing the next best algorithm from the state of the art (ProMNoILP) with a ranking of 4.641. This reinforces the confidence that REACH is much faster, as this rank compares the time to compute alignments for each log-model pair. To confirm again that there are statistically significant differences between REACH and the other algorithms for a time limit of 300 seconds, we have repeated the Friedman's and Holm's tests —based on the ranking from Table 2.9b. Regarding Friedman's test, REACH has the best ranking with a *p-value* lower than 10^{-5}, indicating again that there are statistically significant differences in the performances of the algorithms. Furthermore, Holm's test allows the rejection of the null hypothesis in all the pairwise comparisons between REACH and the other algorithms, since the *p-values* are lower than 10^{-5} in all cases.

2.4.5 Limitations

Although REACH achieves superior performance compared to state-of-the-art algorithms, it still presents certain limitations. REACH does not succeed in solving 11 of the 227 tested log-model pairs due to the nature of the A* search: the number of states to explore explodes based on the complexity of the models and logs, as well as the cost of the optimal alignment. There are several factors that affect the performance of the algorithm in those cases, among which we highlight:

- The parallelism allowed by the process model, i.e., the average number of enabled activities for each reachable marking. Each of those activities must generate a new neighboring state by performing a model move when that marking is reached. Each state added to the search space when exploring the neighbors of a state exponentially increases the time complexity, as each generated state is recursively explored if the search algorithm requests it. This is aggravated by the presence of silent transitions, which allow reaching different markings of the process model without increasing the cost, effectively raising the number of neighboring states. Loops also contribute to this by removing the length limit of paths through the model.

- The length of each trace directly affects the number of moves required in the optimal alignment. As alignments are constructed from start to end, adding a move at each state transition, the depth of the solution in the search space increases with the addition of an event to the trace. Increasing the depth of the explored search space exponentially raises the time complexity of the algorithm.

- The cost of the optimal alignment, i.e., the minimum number of errors that must be repaired for the trace to follow a valid path through the model, forces the A* search to explore more states. This is because the solution will include more asynchronous movements of cost 1, compelling the search algorithm to ensure that there are no complete alignments of lower cost than the optimal one.

Among the 227 tested log-model pairs, REACH was not the fastest optimal algorithm in 100 instances. Notably, these cases primarily correspond to the simplest problems in the experiment. This observation is visually supported by Figure 2.7, where it can be seen that REACH takes slightly longer to solve most of the problems that take less than one second. The median delay

of REACH with respect to the fastest optimal algorithm for each of these pairs is 204ms, and the fastest algorithm is not always the same, varying among ProMNoILP, ProMILP, ProMLP, AutoConf, eMEQ and RecomposingReplay. The delay observed in simple log-model pairs for REACH is primarily attributed to the extended initialization time required by the proposed optimizations. These optimizations have been designed with the aim of enhancing performance in the context of complex log-model pairs.

Even with all optimizations applied, REACH was unable to complete the alignments computation for 11 of the tested log-model pairs within less than five minutes. The only optimal algorithm capable of solving three of these pairs is eMEQ [41]. This algorithm uses a complex heuristic based on Integer Linear Programming (ILP), which proves more effective than REACH at detecting misalignments toward the end of the trace. For each state, the ILP solver can estimate the minimum cost of a solution without overestimating it, thus suiting A* heuristics. Although this estimation is computationally expensive, it is more accurate than REACH , making eMEQ capable of solving three log-model pairs that REACH cannot finish within the given time limit.

2.5 Discussion

This chapter presented REACH, an A*-based algorithm that computes in a very efficient way optimal alignments. The main contributions of this proposal include techniques designed to minimize the number of states explored by the A* algorithm and the utilization of a partial reachability graph for faster execution of process models during alignment computation.

REACH has been tested with 227 log-model pairs from different domains and discovered using different algorithms. The performance of each contribution of the algorithm was verified by partially enabling the proposed optimizations, and the performance of the complete algorithm was compared with 10 state-of-the-art conformance-checking algorithms. Results show that REACH aligns 95% of the tested log-model pairs in less than 45 seconds.

Remarkably, the proposed optimizations empower REACH to complete alignments in just 45 seconds for two log-model pairs that no other algorithm can solve within a 5-minute time frame. Moreover, for a 10-second time limit, it also aligns 26% more pairs than all the other optimal state-of-the-art algorithms. REACH exhibits exceptional speed, outperforming all state-of-the-art approaches by aligning 55% of the log-model pairs more rapidly than any other algorithm.

The performance improvements enable the efficient computation of optimal alignments, allowing users to perform more precise conformance checking. By eliminating the need to rely on fast but less accurate methods, REACH opens up new possibilities for the application of conformance checking in real-world scenarios.

However, REACH still presents some limitations that should be addressed in future work. It currently proposes a balanced heuristic between accuracy and computing time, but this may not return the fastest solution for some log-model pairs: (i) for some high-complexity pairs, Integer Linear Programming (ILP) can be used to define more accurate heuristics (although slower) than our heuristic, or (ii) very easy to compute optimal alignments, that could be solved faster by algorithms that compute simple heuristics very quickly, albeit inaccurately. The SRLog optimization also slightly increases the computation time for the simplest problems due to the relatively slow initialization phase.

Therefore, as future work, we plan to develop a more advanced heuristic based on ILP, focusing on reducing the time spent on each state. Nevertheless, simpler problems are solved faster when using simpler heuristics and disabling the SRLog optimization. Hence, we will propose a new classification technique that, based on the characteristics of the input log and model, selects the heuristic and optimizations that best tackle the given problem.

Chapter 3

DeclareAligner: A Leap Towards Efficient Optimal Alignments for Declarative Process Model Conformance Checking

The contents of this chapter are partially extracted from the following publication:

Publication

Jacobo Casas-Ramos[a], Manuel Lama[a], and Manuel Mucientes[a]. DeclareAligner: A Leap Towards Efficient Optimal Alignments for Declarative Process Model Conformance Checking. *Engineering Applications of Artificial Intelligence*, 2025.
DOI: 10.1016/j.engappai.2025.111683

Journal Impact Factor (JCR 2024): 8.0

- Computer Science, Artificial Intelligence: **Q1** (25/204)

[a]CiTIUS, Universidade de Santiago de Compostela, Santiago de Compostela, Spain.

In the previous chapter, we presented REACH, an efficient algorithm for computing optimal alignments between event logs and procedural process models. This chapter shifts the focus to aligning declarative process models, which are better suited for dynamic business processes

due to their flexibility.

Unlike procedural models, declarative models do not explicitly define the control flow. Essentially, declarative models specify *what* constraints must be satisfied during process execution without prescribing exactly *how* it should be done [26]. The DECLARE language [43] is the leading approach to declarative process modeling, offering a flexible and concise way to define constraints on business processes by filling constraint templates with relevant activities. It enables analysts to define permissible behavior using constraints rather than enumerating every possible execution path. This approach allows for a more abstract and high-level representation of business processes, making it easier to model complex and dynamic behaviors.

The flexibility allowed by DECLARE process models introduces unique performance challenges in conformance checking, particularly when computing optimal alignments. This is due to the permissive nature of declarative models, which allows for a large number of possible execution paths. As a result, the search space of possible alignments becomes significantly larger, making it more difficult to efficiently compute optimal alignments.

Despite recent advancements, existing approaches to computing optimal alignments for declarative conformance checking often struggle with scalability and efficiency, limiting their applicability in real-world settings. Therefore, developing new algorithms and techniques that can efficiently handle the complexity of declarative process models is necessary for improving the effectiveness of conformance checking in dynamic business processes.

State-of-the-art approaches [44, 45, 46] translate the DECLARE language into Linear-time Temporal Logic over finite traces (LTLf) formulas and then into finite-state automata [47]. These automata can then be used to compute alignments using algorithms similar to those utilized for procedural models. This translation process can lead to a loss of semantic simplicity and an increase in computational complexity. Specifically, the translation process can obscure the declarative nature of the original constraints, thereby forfeiting potential optimizations that could be leveraged if the declarative representation were used directly for computing optimal alignments.

This chapter presents DECLAREALIGNER, a novel algorithm that addresses the challenge of efficiently computing optimal alignments for declarative process models. It extends the A* search algorithm to tackle the problem from a fresh perspective, leveraging the flexibility of declarative models by using them directly during the alignment process. A* is renowned for its ability to find the shortest path between two nodes in a graph. In the context of DECLAREALIGNER, these nodes are referred to as states, and the graph —which we term

the search-space— is constructed incrementally as it is explored by the A* algorithm. The proposed search space is initialized with a starting state that embodies the entire trace. From this foundation, new states are iteratively generated through the application of targeted corrective actions. As these actions are applied, each successive state exhibits increasing compliance with the process model, ultimately yielding a goal state that corresponds to a valid execution of the process. By comparing the original trace to a valid trace represented by this goal state, an optimal alignment can be extracted, highlighting the deviations between the actual and expected behavior. Our approach offers several significant innovations that enhance its performance and scalability:

1. It only considers taking actions that directly contribute to repairing violated constraints.
2. It proposes several search optimization strategies in order to manage difficult alignments:
    a) An heuristic that merges the suggested actions for all violated constraints of a state to provide accurate cost estimates.
    b) States that cannot reach the optimal alignment are pruned early to eliminate unfruitful branches of the search space.
    c) Actions that are required by chain constraints and do not interfere with other constraints are applied before the search.
    d) Actions perform multiple operations at once to avoid intermediate states.

The remainder of this chapter is organized as follows. Section 3.1 reviews existing approaches and highlights their limitations. Section 3.2 introduces the necessary concepts used by the algorithm. Section 3.3 presents the DeclareAligner algorithm, including its key components and optimizations. Section 3.4 discusses the evaluation of DeclareAligner, highlighting its performance and scalability advantages over existing state-of-the-art approaches. Finally, Section 3.5 concludes the chapter by summarizing the main contributions and outlining potential directions for further investigation.

3.1 Related work

Declarative conformance checking, particularly using the DECLARE language [43], has gathered significant attention in recent years due to its ability to model complex and flexible business

processes and reason about their behavior. Several approaches have been proposed for declarative conformance checking. These can be broadly categorized into two types: those that report only basic conformance information and those that compute optimal alignments to provide detailed insights into process deviations.

Methods in the first category, such as [105], simply execute recorded events in the process model and return a boolean result indicating conformance or non-conformance, without providing insights into specific issues. Other methods in this category, including [48, 106, 107, 108, 109, 110] report activations, satisfactions, and violations for each constraint within the DECLARE model and for every trace in the input log. While these results are faster to compute than optimal alignments, they do not identify root causes of non-conformance or suggest fixes. The main limitation of these approaches is that they only provide a high-level indication of conformance or non-conformance, without offering actionable insights for process improvement.

In contrast to simpler conformance checking methods, computing optimal alignments enables a more detailed analysis of process deviations and identification of opportunities for improvement. However, these approaches come at a higher computational cost. Currently, there are only two state-of-the-art approaches for computing optimal alignments of declarative models, namely those presented in [44] and [54]. A key characteristic shared by these approaches is the initial conversion of the DECLARE process model to finite-state automata. Specifically, standard DECLARE constraints can be translated into Linear-time Temporal Logic over finite traces (LTLf) specifications, which in turn can be converted into finite-state automata for each constraint [111, 112]. The two state-of-the-art approaches differ primarily in their techniques for searching for the optimal alignment using these automata:

- The first approach [44] employs the A* algorithm to find optimal alignments, providing metrics such as fitness, precision, and generalization. However, this method can be computationally expensive due to the need to explore a large state space using a relatively simple heuristic.

- The second approach [54] converts automata to a planning problem and uses classical planners to find optimal alignments. This approach has shown better efficiency but may face scalability issues due to the required conversion to a planning task or the lack of control over the approach that the planner takes.

The field of declarative conformance checking is continually evolving, with researchers exploring novel approaches to enhance its capabilities. For example, [51] proposes a method that leverages SAT solvers for data-aware declarative process mining, introducing an additional layer of complexity by incorporating data-aware constraints into the alignment problem. Similarly, [28] investigates the application of planning techniques for aligning data-aware declarative process models, which further complicates alignment computation by considering the impact of data attributes on process conformance. Additionally, [52] contributes to the understanding of efficient optimal alignment computation, although its focus on DCR Graphs may limit its broader applicability, as DCR Graphs are not as widely adopted as DECLARE. Moreover, [45] contributes to this area by examining the conformance checking of mixed-paradigm process models. However, it is noted that when such models contain solely declarative constraints without procedural constructs, they generate the largest possible search space, leading to performance degradation of the A* algorithm. The growing sophistication of these approaches creates a corresponding need to navigate increasingly vast search spaces, making efficient alignment algorithms for declarative process models a pressing requirement.

Existing methods for computing optimal alignments using DECLARE models struggle with large process models and logs. The proposed DeclareAligner algorithm addresses these limitations by introducing a novel approach that focuses only on actions that can potentially resolve violated constraints, preprocesses the problem to reduce unnecessary work, and groups multiple fixes together into single actions to minimize redundant effort. Additionally, the algorithm utilizes an advanced heuristic and detects and removes dead-ends from the search space. This results in improved performance and scalability, enabling the tackling of more complex optimal alignment tasks.

3.2 Preliminaries

Conformance checking requires two key components: a log and a process model. A log is comprised of events, which are organized into traces that represent individual executions of the process model. Each event within a trace contains essential information, including its execution timestamp, a unique identifier linking it to other events in the same process execution (trace identifier), and the specific activity executed. For simplicity, a trace $\sigma = \langle A_1, \ldots, A_n \rangle$ is defined as a sequence of activities ordered by their execution timestamps, as formally outlined in Definitions 1.1 to 1.3. An example log is shown in Table 3.1.

Table 3.1: Event log data collected for an e-learning process. Each row shows an event. An example trace with ID *Case234* is ⟨*Enroll*, *Test*, *Exam*⟩.

Trace ID	Activity	Timestamp
Case234	Enroll	2023-09-01 22:16:29
Case675	Class	2023-11-01 04:10:07
Case234	Test	2023-11-15 02:09:48
Case675	Exam	2023-12-13 07:24:41
Case234	Exam	2024-01-19 06:42:33
⋮	⋮	⋮

The most widely used declarative process modeling language is DECLARE [43]. It defines a list of parametrized templates that, when instantiated with specific activities, become constraints restricting the execution of those activities. Table 3.2 provides a comprehensive overview of all supported DECLARE templates. For formal definitions, refer to [113]. A DECLARE process model is simply a set of constraints.

A constraint is considered violated when the specified condition or rule is not met within the trace. This means that the activities in the process occur in a manner that breaches the constraint. If a constraint is not violated, it is considered satisfied. A few constraints such as `Init`(A) are always active. However, most constraints have an activity that behaves as the activation. If the activation does not appear in the trace, the constraint is always (vacuously) satisfied. For example, the activity A of `Response`(A, B) is the activation, as this constraint ensures that B must appear at some point after A, only if A appears in the trace.

Branching is an important part of the DECLARE language as it allows defining more complex constraints by inserting multiple activities instead of just one in each parameter of the template. When multiple branched activities are applied to a parameter of a template, any of them can trigger the effect associated with the parameter. Branched activities are shown between square brackets to avoid confusion.

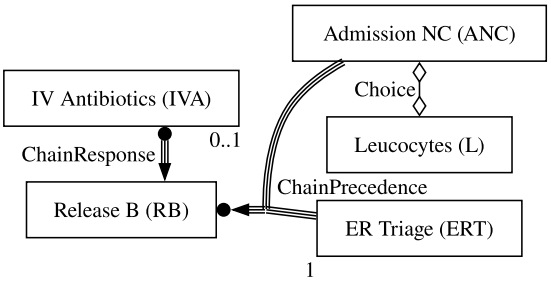
Table 3.2: Supported DECLARE constraints and their natural language descriptions.

Existence(n, A): A occurs at least n times.	*Participation(A)*: A occurs at least once.
Absence(n, A): A occurs at most n-1 times.	*AtMostOne(A)*: A occurs at most once.
Exactly(n, A): A occurs exactly n times.	*Init(A)*: A is the first activity.
End(A): A is the last activity.	*Choice(A, B)*: Either A or B, or both, occur.
ExclusiveChoice(A, B): Either A or B, but not both, occur.	*RespondedExistence(A, B)*: If A occurs, B also occurs.
Response(A, B): If A occurs, B follows.	*Precedence(A, B)*: If B occurs, A precedes it.
AlternateResponse(A, B): If A occurs, B follows without any other A in between.	*AlternatePrecedence(A, B)*: If B occurs, A precedes it without any other B in between.
ChainResponse(A, B): If A occurs, B is the next activity.	*ChainPrecedence(A, B)*: If B occurs, A is the previous activity.
CoExistence(A, B): If A occurs, B also occurs, and vice versa.	*Succession(A, B)*: Combines Response and Precedence.
AlternateSuccession(A, B): Combines AlternateResponse and AlternatePrecedence.	*ChainSuccession(A, B)*: Combines ChainResponse and ChainPrecedence.
NotRespondedExistence(A, B): If A occurs, B does not.	*NotCoExistence(A, B)*: If A occurs, B does not, and vice versa.
NotResponse(A, B): If A occurs, B does follow.	*NotPrecedence(A, B)*: If B occurs, A does not precede it.
NotSuccession(A, B): If A occurs, B does not follow, and if B occurs, A does not precede it.	*NotChainResponse(A, B)*: A is not immediately followed by B.
NotChainPrecedence(A, B): B is not immediately preceded by A.	*NotChainSuccession(A, B)*: A is not immediately followed by B and B is not immediately preceded by A.

Example 3.1: Example of a DECLARE process from a hospital.

Figure 3.1 shows an example model from a hospital in both graphical and textual form with the following constraints:

- `Exactly`(1, *ERT*) specifies that *ERT* must occur exactly once in each trace.



(a) Graphical representation.

Choice(ANC, L)
ChainPrecedence([ERT, ANC], RB)
Absence(2, IVA)
Exactly(1, ERT)
ChainResponse(IVA, RB)

(b) Text version.

Figure 3.1: Example of a DECLARE process from a hospital.

- `Absence`(2, *IVA*) indicates that *IVA* cannot occur more than once.
- `Choice`(*ANC*, *L*) defines that either *ANC* or *L* must appear at some point in the trace.
- `ChainResponse`(*IVA*, *RB*) enforces that whenever *IVA* occurs, *RB* must also occur immediately after it.
- `ChainPrecedence`([*ANC*, *ERT*], *RB*) is an example of branching that specifies that whenever *RB* occurs, *ANC* or *ERT* must occur immediately before it.

An alignment maps a trace to a process model, highlighting conformance issues and enabling applications like model repair and auditing. An alignment is a sequence of legal moves that traverse both the trace and the process model from start to finish.

Definition 3.1: Legal move (DECLARE)

Let M be a DECLARE model with activities M, L be a log with activities A_L, and i be the first index (zero-based) of the trace that still needs to be aligned such that $\sigma[i]$ is the next activity to align. A move is a tuple (a_L, a_M), where $a_L \in A_L \cup \gg$ and $a_M \in A_M \cup \gg$. Legal moves are moves of the following forms:

- **Synchronous moves** $(\sigma[i], \sigma[i])$ are available if the execution of the activity a_n does not permanently violate any constraint of M. This updates the state of the model accordingly and advances the index i to the next trace activity.

- **Log moves** $(\sigma[i], \gg)$ are available if $i < |\sigma|$. This move increments i by one, progressing in the trace without affecting the model.
- **Model moves** $(\gg, a)$ where $a \in A_M$ are available if all constraints of M would not become permanently violated if a is executed. This updates the state of the model by executing a.

Log and model moves may be referred to as asynchronous moves.

Definition 3.2: Alignment (DECLARE)

An alignment $\gamma = \langle \gamma_1, \ldots, \gamma_n \rangle$ is a sequence of legal moves which reaches the end of the trace and a state of the model which satisfies all constraints.

A cost function assigns a penalty to each move in an alignment based on how much the observed behavior deviates from the modeled behavior. The default cost function, used consistently throughout this chapter, applies a cost of 1 to asynchronous moves, while synchronous moves incur no cost (i.e., cost of 0).

Definition 3.3: Alignment cost and optimal alignment (DECLARE)

A cost function $C : (A_L \cup \{\gg\}) \times (A_M \cup \{\gg\}) \setminus (A_L \times A_M \cup \{\gg\} \times \{\gg\}) \to (0, \infty)$ assigns a cost c_{γ_i} to each possible legal asynchronous move, where A_L is the set of activities in the trace and A_M is the set of activities in the DECLARE model. Each valid alignment is assigned a cost $c_\gamma = c_{\gamma_1} + \cdots + c_{\gamma_n}$. An optimal alignment for a given trace and model is any of the alignments with the minimum total cost.

Example 3.2: Alignment for the hospital process

Table 3.3 shows an example alignment for the hospital process, where each legal move is represented as a column. The top and bottom rows correspond to the log and model components, respectively. This particular alignment has a total cost of 2 and is one of the optimal alignments for this example. Notably, the asynchronous moves in this alignment indicate that the ER Triage (ERT) activity should have been performed instead of the IV Antibiotics (IVA) activity for this specific trace, highlighting a deviation from the

> expected process behavior along with its most likely repair path.

Table 3.3: Alignment of the trace ⟨ANC, L, IVA, RB⟩ with the process depicted in Figure 3.1. Activities are represented by their initials.

Log:	ANC	L	≫	IVA	RB
Model:	ANC	L	ERT	≫	RB

3.3 DeclareAligner algorithm

DeclareAligner is built on top of A* search [33], a popular pathfinding and graph traversal algorithm that efficiently finds the shortest path between two nodes in a weighted graph. The A* search algorithm, known for its completeness and optimality guarantees, is particularly well-suited for the optimal alignments task as this task can be expressed as a graph that repairs violated constraints until all constraints are satisfied. The nodes of this graph are referred to as states.

The DeclareAligner algorithm consists of several key components, as illustrated in Figure 3.2. The process commences with an initial state, which represents the current trace (Section 3.3.1). Prior to initiating the search for the optimal alignment, the algorithm performs preprocessing on the initial state (Section 3.3.5.2) effectively reducing the workload by starting closer to the solution. Subsequently, the A* search algorithm recursively explores a search space that is tailored to the computation of optimal alignments. The graph is incrementally constructed by applying fixes to violated constraints, thereby generating neighboring states (Section 3.3.2). This exploration is facilitated by various optimizations, including a tailored heuristic (Section 3.3.3), early pruning of dead-ends (Section 3.3.5.1), and the grouping of multiple fixes into single actions (Section 3.3.5.3). When a state that has no violated constraints is explored, the optimal alignment can be extracted (Section 3.3.4).

3.3.1 State definition and notation

The algorithm explores a search space where each state is created by applying fixes to resolve constraint violations. A state contains the following attributes:

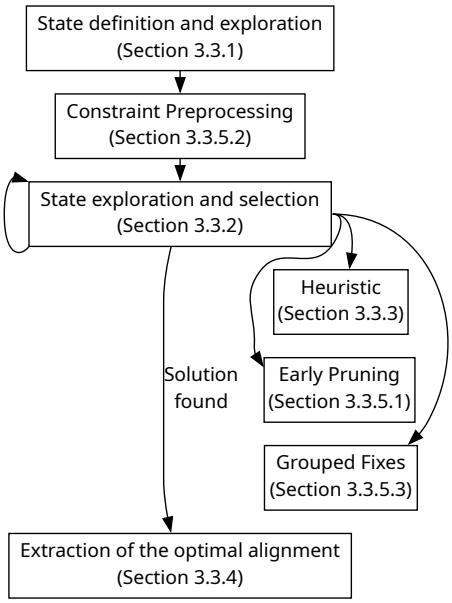


Figure 3.2: High-level overview of DeclareAligner.

- **Cost** is a measure of the severity of fixes required to reach the state from the initial state.
- **Heuristic** is an estimate of the additional cost needed to transition from the current state to a goal state where all constraints are satisfied.
- **LTGraph** is a directed acyclic graph showing dependencies between activities or activity groups, with nodes connected by arcs that indicate precedence relationships. Activity groups can represent two types of relationships: branched activities, denoted as [A, B], where all activities have the same effect so they are interchangeable; or chained activity groups, denoted as A > B, where B directly follows A.
- **Violated activations** is a list of constraint activations that remain violated.
- **Will fix** is the activation that will be addressed next (Section 3.3.2).

The algorithm commences with an initial state, denoted as *state0*, which serves as the starting point for the search process. The initial state is constructed by creating an LTGraph that reflects the original problem instance: each activity of the trace is added as a node and

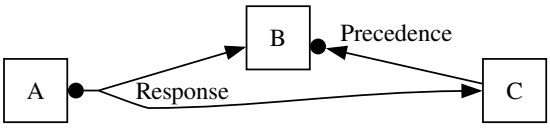


Figure 3.3: Process model of the running example.

connected to the node for the next activity of the trace to indicate their strict precedence. The initial state has a cost of 0 since no edits have been made yet.

Example 3.3: Running example for the algorithm section

To facilitate understanding of the concepts presented, a running example will be utilized throughout this section. The running example consists of the trace $\langle A, A \rangle$ and the process model shown in Figure 3.3:

- `Response`(A, [B, C]) enforces that if A appears in the trace (activation), then either B or C must occur at any point after the activation.
- `Precedence`(C, B) specifies that if B appears in the trace (activation), then C must occur at any point before the activation.

The initial state for the running example is depicted in Figure 3.4, which shows from top to bottom:

- State identifier (*State0*), cost and heuristic values. The cost of the initial state is always 0, whereas the heuristic value is computed by identifying actions that are required to reach a goal state and adding their costs (Section 3.3.3).
- The LTGraph, which enforces the order between the two A activities present in the trace. Distinct subscripts identify each activity instance to avoid ambiguity.
- Lists of violated activations for each process model constraint. The initial state reveals that A_0 and A_1 are violated activations of the response constraint, stemming from the absence of B or C activities succeeding each A activity in the LTGraph. There are no violated activations for the precedence constraint, as there is no B in the LTGraph.

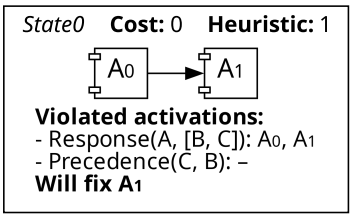


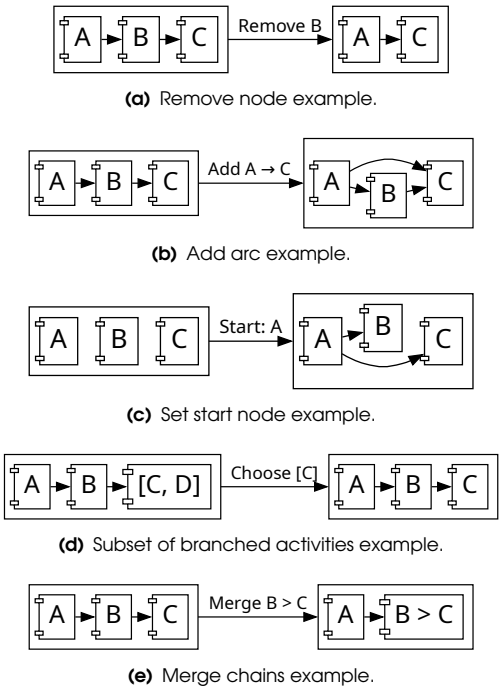
Figure 3.4: Initial state for the running example.

- The selected activation to repair, which in this case is A_1 (Section 3.3.2).

3.3.2 State exploration and selection

Neighbor generation is guided by identifying violated constraint activations and proposing repair actions. When multiple violations occur, those suggesting actions with the highest average costs are prioritized. This strategy of exploring high-cost actions first is grounded in the observation that non-zero cost actions result in asynchronous moves in the optimal alignment (Definition 3.1). Such actions can have a profound impact on other constraints, as they involve adding and removing activities. By prioritizing high-cost actions early in the search process, the risk of cascading effects that can arise when they are applied deeper in the search space, where numerous states are awaiting exploration, is mitigated. As a result of resolving high-cost actions first, subsequent repair actions suggested by other constraints tend to have localized effects on the state. This makes them more likely to be successfully repaired without triggering additional conflicts, and ultimately facilitating a faster convergence towards an optimal alignment.

In cases where multiple violated activations have equal average costs, a secondary prioritization strategy to resolve ties is employed. Forward-looking constraints —e.g., `Response`— prioritize their last failed activation because all activations can be fixed if the target is inserted after this point. Analogously, backward-looking constraints favor their first failed activation. Negative constraints, however, require the opposite approach: since the goal is to avoid introducing targets altogether, removing the targets from the first activation effectively fixes all subsequent activations for forward-looking constraints and removing them from the last activation does so for backward-looking ones.



(a) Remove node example.

(b) Add arc example.

(c) Set start node example.

(d) Subset of branched activities example.

(e) Merge chains example.

Figure 3.5: Examples of fix kinds applied to an LTGraph.

Generating neighboring states is accomplished by constructing actions, which comprise sequences of corrective modifications applied to the LTGraph in response to a violated constraint activation. All violations of DECLARE constraints can be repaired by applying a combination of the following fix kinds:

- **Insert or remove node (Fig. 3.5(a))**. Adds or deletes a node of the LTGraph, updating adjacent arcs when removing a node.
- **Add an arc (Fig. 3.5(b))**. Enforces the origin node to precede the destination node.
- **Set start or end node (Fig. 3.5(c))**. Sets a node as the first or last node, adding arcs to all other nodes and ensuring it remains in that position.

- **Subset of branched activities (Fig. 3.5(d))**. In scenarios where a constraint is activated or violated by only a subset of branched activities present in a LTGraph node, this fix kind can be employed to avoid activation or enforce the target, respectively.
- **Merge or split chains (Fig. 3.5(e))**. To efficiently implement chain constraints, the LTGraph nodes are capable of representing lists of possibly branched activities (separated by >). This fix kind merges two nodes, preserving their chained relationship without the need to add arcs to all other nodes.

Only the insert or remove node fix kind has a non-null cost. This is because adding a node triggers a model move in the optimal alignment, while removing a node causes a log move (Section 3.3.4). In contrast, all other fix kinds only reduce the set of possible alignments that the LTGraph represents, without modifying their costs.

The standard cost function assigns a uniform cost of one to each insertion or removal operation. The cost of an action is the sum of the costs of its individual fixes. Each action taken contributes to increasing the cost of the current state.

Definition 3.4: State cost

Let T be the complete set of actions applied to the initial state in order to reach the current state s, and let $ct(t)$ be the cost of the action t, the cost of s can be defined as:

$$cost(s) = \sum_{action \in T} ct(action)$$

The next state to explore is the previously unexplored state with the lowest total estimated cost ($cost + heuristic$). The algorithm continues to iteratively explore and select states until either a goal state is explored or no further states remain to be explored, in which case there is no optimal alignment.

Example 3.4: Search space for the running example

Figure 3.6 illustrates the complete search space explored by the algorithm to find the optimal alignment for the running example. The representation of an action is an arc connecting the parent state to the neighboring state. The arc is labelled with the fixes of the action. Goal states are highlighted with a green border, and the path to the optimal

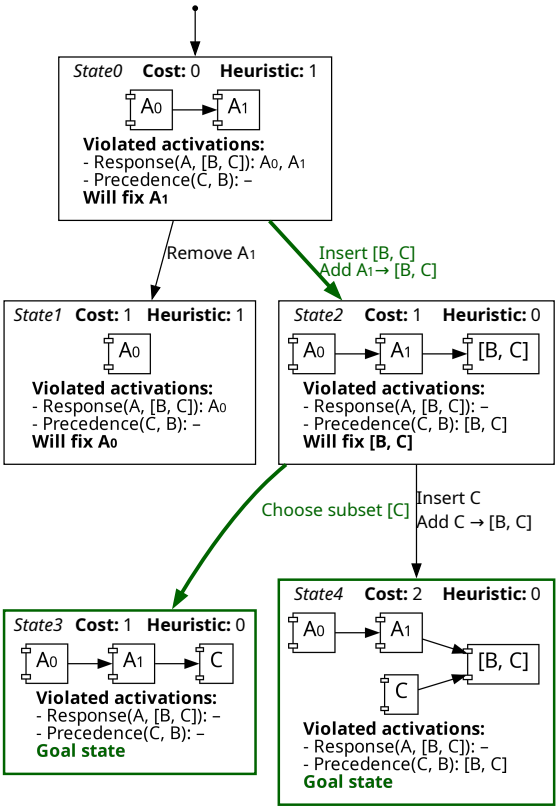


Figure 3.6: Search space discovered for the running example.

goal state is marked with green arrows.

As previously mentioned the initial state (*state0*) has two violated activations for the response constraint. In this case, the algorithm selects the A_1 activation from the LTGraph to repair. There are only two possible fixes, which create the neighboring states of *state1* and *state2*:

- *State1* removes the activation so that the constraint is never activated, with a cost of 1.
- *State2* inserts the expected target of the constraint, which can be either of the

activities B or C. This results in the addition of the LTGraph node $[B, C]$ after the activation. This action results in satisfying both activations of the response constraint. However, it also activates the precedence constraint as one of the branched activities triggers it.

The algorithm selects *state2* to explore next based on its lower $cost + heuristic$ value. *state2* has only one violated activation to repair for the precedence constraint, resulting in the generation of the following neighboring states:

- *State3* removes the activation through a cost-free operation, which involves selecting a subset of branched activities to avoid triggering the precedence constraint.
- *State4* inserts the target activity before the activation. Note that the LTGraph now aggregates a series of possible activity orderings into a single state, reducing the size of the search space.

Ultimately, the algorithm reaches the goal state with the lowest cost, which in this case is *state3*. The optimal alignment can then be extracted from the goal state (Section 3.3.4).

3.3.3 Heuristic

The heuristic function used in DeclareAligner leverages knowledge about required repair actions to provide an accurate estimate of the remaining cost to reach a goal state. Given that all violated activations must be addressed, this approach is based on two key insights:

- **Minimum required actions**. At least one action from the set of repair actions suggested for each violated activation is necessary to resolve the violation.
- **Optimistic merging**. Actions that can fix multiple violations must be merged optimistically to avoid overestimating the remaining cost.

The heuristic function, formally defined in Alg. 3.1, takes advantage of these insights to compute an accurate estimate of the remaining cost. The process begins by identifying all

Algorithm 3.1: Heuristic function

```
1:  function HEURISTIC(state)
2:      activations ← violatedActivations(state)
3:      actionSets ← {actions(a) | a ∈ activations}
4:      combination ← HDIJKSTRA(actionSets)
5:      return Σ_{action∈combination} ct(action)
6:  end function
7:  function HDIJKSTRA(actionSets)
8:      hstates ← {{}}                          ▷ Set of unexplored hstates
9:      loop                                    ▷ Main loop of Dijkstra's algorithm
10:         hstate ← REMOVELEASTCOST(hstates)
11:         if hstate is a goal state then
12:             return hstate
13:         hactions←HACTIONS(hstate, actionSets)
14:         for all haction ∈ hactions do
15:             child ← HEXPLORE(hstate, haction)
16:             ADDORREPLACE(hstates, child)
17:         end for
18:     end loop
19: end function
```

violated activations in the current state (alg. 3.1:2). For each violated activation, the set of proposed repair actions is retrieved (alg. 3.1:3).

Next, Dijkstra's algorithm [114] is used to search for the combination of one action from each set that incurs the lowest total cost (alg. 3.1:4). The main loop of the search (alg. 3.1:9) involves four key functions:

- The REMOVELEASTCOST function (alg. 3.1:10) deletes and returns the unexplored heuristic state with the lowest total cost from the set of *hstates*, where the cost is determined by the sum of action costs. This state becomes the next candidate for exploration in the search process.
- The HACTIONS function (alg. 3.1:13) selects the next violated activation that was not previously explored and adds all of its suggested fixes as new actions to discover new neighboring states.
- The HEXPLORE function (alg. 3.1:15) discovers a new *child* state from each of the proposed actions of the HACTIONS function. It does so by adding the selected action to

the current state and merging it with existing actions to avoid overestimating the cost.

- The AddOrReplace function (Alg. 3.1:16) checks whether the generated *child* state is already in *hstates*. If not, it adds the $child$; otherwise, it replaces the existing *hstate* only if the new one has a lower cost.

The search algorithm concludes upon exploring the first *hstate* that meets the goal criteria, namely, when the *hstate* includes at least one action from each activation (alg. 3.1:11). This is always found since any combination that includes actions from all activations is valid, and Dijkstra's algorithm ensures that the optimal one is found (alg. 3.1:12). Hence, the returned *hstate* is the combination of actions from all violated activations that yields the lowest cost. The resulting heuristic value is determined by this minimum possible cost of the combined actions (alg. 3.1:5). This value provides an informed estimate of the effort required to resolve all outstanding issues and achieve a valid solution.

Example 3.5: Heuristic computation for the running example

Consider *state0* from Figure 3.6. Two similar violated activations propose actions that either remove the activation or insert a new node after it, both of cost 1. However, the insert actions can be combined to solve both violations simultaneously. The heuristic searches for the lowest-cost combination of actions, which in this case is a single insert action of cost 1. This yields a heuristic value of 1 for *state0*.

3.3.4 Extraction of the optimal alignment

To extract the optimal alignment from the goal state, a topological sort for the LTGraph is found. A topological sort is a linear ordering of nodes in the LTGraph such that for every edge, the starting node comes before the ending node in the order. There can be multiple topological sortings for the nodes of an LTGraph, but the algorithm ensures that any one of them will have equivalent cost for a goal state. Since the objective is to obtain any optimal alignment, selecting any one of them results in the optimal alignment of the problem. This results in a sequence of activities executed in the model, where branched activities are resolved by selecting any one of them.

We then process both the nodes from the topological sort and the original trace simultaneously to extract the optimal alignment:

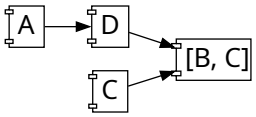


Figure 3.7: Example of an LTGRAPH with branched activities.

- If an LTGRAPH node was inserted when performing an action, a model move is added to the alignment and advance the topological sort.
- Otherwise:
  - If the next LTGRAPH node matches the next activity in the trace, a synchronous move is added to the alignment and advance both.
  - Otherwise, a log move is added to the alignment and advance the trace.

Any remaining activities in the trace are handled by adding them as log moves. The collected sequence of moves is the optimal alignment.

Example 3.6: Extraction of the optimal alignment from an LTGRAPH

Let $\langle A, D, A \rangle$ be the trace and Figure 3.7 be the LTGRAPH of the goal state from which the optimal alignment will be extracted. A valid topological sort of the LTGRAPH is $\langle A, C, D, C \rangle$, after selecting C from the branched activities $[B, C]$. This yields an optimal alignment where A is a synchronous move, followed by a model move for the inserted activity C, another synchronous move for D, and another model move for the inserted activity C. Finally, since the last A from the trace was not processed, it results in a log move, producing the optimal alignment shown in Table 3.4.

Table 3.4: Example of optimal alignment extracted from the LTGRAPH shown in Figure 3.7 and the trace $\langle A_0, D, A_1 \rangle$.

LOG	A	≫	D	≫	A
MODEL	A	C	D	C	≫

3.3.5 Optimizations

Several optimization techniques can further improve the efficiency and effectiveness of the DeclareAligner algorithm. These enhancements build upon the foundation established in the previous sections and are designed to refine the performance of the algorithm.

Early Pruning

The DeclareAligner algorithm improves search efficiency by pruning branches that cannot lead to a goal state. If any violated activation has no repair action, it is impossible to reach a goal state from that point. The reason behind this is that the violated activation will never be repaired even if all other violated activations are eventually repaired through the exploration of the search space. Since the heuristic already computes the suggested actions for all violated activations, the detection of dead-ends incurs no additional overhead.

To determine whether an action is feasible for a given state, a cycle detection algorithm is executed on the resulting LTGraph after applying the action. This is because the LTGraph represents a strict order, and any cycles would result in an impossible topological sort. In other words, if a cycle were present, it would indicate that there are conflicting requirements in the graph, making it impossible to extract a valid alignment from the given state. By detecting such cycles, actions that would lead to inconsistencies can be identified and pruned.

Example 3.7: Early Pruning

The example illustrated in Figure 3.8 demonstrates the benefits of early pruning. Initially, the end constraint is repaired by inserting a B activity at the end of the LTGraph. Upon exploring this second state, it is identified that the alternate succession constraint would require the insertion of a C after the B, and this cannot be satisfied without creating a cycle in the LTGraph ($B \rightleftarrows C$), indicating conflicting requirements. As a result, this state is recognized as a dead-end, since repairing the alternate succession constraint is necessary to reach a goal state.

Without early pruning, the algorithm would proceed by addressing the violated A_0 activation of the init constraint, leading to the discovery of numerous additional neighbors before ultimately realizing that each of these states cannot reach a goal state due to the impossibility of solving the alternate succession constraint. In this simple

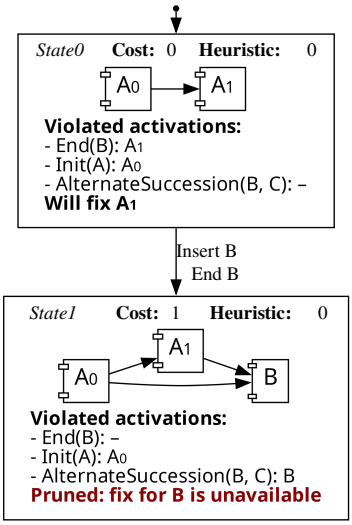


Figure 3.8: Illustrative example highlighting the advantages of enabling the Early Pruning optimization, cutting down the search space from 7 discovered states to just 2.

> scenario, enabling early pruning reduces the search space from 7 states to 2 states. For problems with more constraints, the difference is even more pronounced, as checking all constraints for dead-ends prevents unnecessary exploration and avoids realizing later that no solution exists.

Constraint Preprocessing

The DeclareAligner algorithm can leverage preprocessing techniques to reduce problem complexity for certain constraints, specifically those containing *chain* in their name. By merging nodes in the LTGraph of the initial state that are affected by chain constraints, the algorithm can significantly improve efficiency while maintaining optimality. To achieve this, care must be taken to ensure that merged nodes can be split later if necessary.

This preprocessing step involves combining two consecutive chained activities into a single node in the initial state if they satisfy the constraint. By applying this preprocessing step before initiating the search, a substantial number of states can be eliminated from consideration, resulting in improved overall efficiency.

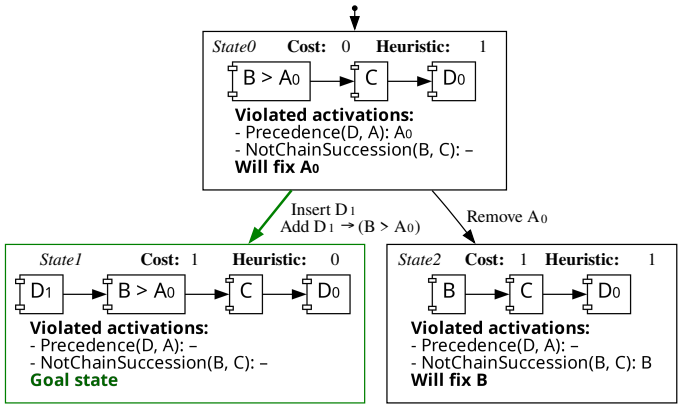


Figure 3.9: This example shows how Constraint Preprocessing simplifies the search space by decreasing the number of discovered states from 9 to just 3.

Example 3.8: Constraint Preprocessing

Figure 3.9 illustrates the impact of this optimization on the search space. In this example, the process model requires that immediately after an B, a C does not appear. Since this condition is met in the initial trace, the optimization merges the nodes for B and A_0 into a single LTGraph node.

If Constraint Preprocessing were disabled, the nodes of the LTGraph would still need to be merged to enforce the not chain succession constraint, necessitating an intermediate action. Consequently, the search space for this simple example would consist of 9 states instead of just 3 states. Notably, this optimization enables chain constraints to behave similarly to most other constraints: if there is no issue with the original trace, there is no need to generate repair actions during the search.

Grouped Fixes

This optimization technique involves consolidating multiple fixes into a single action, thereby eliminating the need to propose individual fixes separately. This approach is particularly effective when dealing with complex actions that require a specific sequence of fixes. For instance, in cases where an activity must appear a certain number of times in the trace,

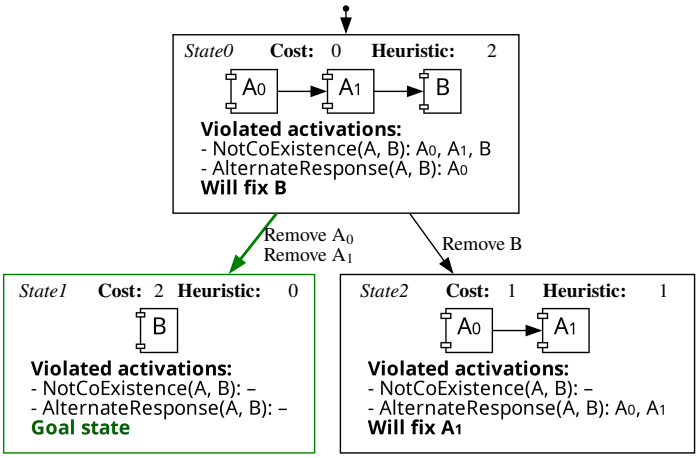


Figure 3.10: Example search space demonstrating the benefits of the Grouped Fixes optimization, which reduces the number of discovered states from 7 to 3.

grouping the necessary fixes into a single action allows for efficient insertion of all the required occurrences simultaneously.

Consolidating these fixes has a significant impact on the search space. By avoiding the generation of intermediate states, the algorithm reduces the need to explore many more unnecessary combinations of fixes.

Example 3.9: Grouped Fixes

Figure 3.10 illustrates the benefits of this optimization technique. Consider a scenario where the `NotCoExistence`(A, B) constraint is violated because both A and B appear in the trace. In this case, the only possible actions to resolve the violation are removing all instances of A or removing all instances of B. Thanks to grouped fixes, these actions can be performed in a single exploration step, allowing the algorithm to reach the goal state after discovering only 3 states.

In contrast, disabling this optimization would result in a significantly larger search space. For this simple example, the search space would consist of 7 states instead of just 3, highlighting the effectiveness of grouped fixes in improving the efficiency of the algorithm.

3.4 Evaluation

DeclareAligner is implemented in Kotlin and executed on the same Java Virtual Machine[1] as the other state-of-the-art algorithms that have been tested. Experiments are conducted on an Intel Xeon Gold 5220R CPU in single-threaded mode with 4GiB of RAM. The source code, dataset and binaries are available online[2].

3.4.1 Datasets

The evaluation utilizes the same datasets as those in [46], extended with additional alignment problems to include all DECLARE constraint templates.

- **D1. Real-life dataset from [46]**. A personal loan application process log from a Dutch financial institute serves as the real-life dataset. Its process model contains 16 constraints and the dataset has a total of 854 trace-model pairs.
- **D2. Synthetic dataset from [46]**. Synthetic logs are generated using three DECLARE models with 10, 15, and 20 constraints. To introduce noise into the system, a modification strategy wherein a subset of constraints are replaced with their negative counterparts was employed. Specifically, 3, 4, or 6 constraints are randomly selected and substituted with their negated versions, while maintaining the same activities as arguments. For instance, the `RespondedExistence`(A, B) constraint may be replaced with its negation, `NotRespondedExistence`(A, B). The log generator from [115] produces four logs for each modified model, containing 100 traces of varying lengths (1-50, 51-100, 101-150, and 151-200 events). These noisy logs are aligned with the original models, totaling 3,600 trace-model pairs.
- **D3. Extended dataset**. 16 DECLARE templates were missing from the original test datasets, including alternate relation constraints, choice constraints, and various negative constraints. Additional synthetic log-model pairs are generated following the same procedure described in the previous paragraph, but ensuring all constraint templates are present in the results. This adds another 3,600 trace-model pairs.

The complete dataset contains 8,054 trace-model pairs.

[1] OpenJDK Temurin-22.0.1+8.

[2] `https://apps.citius.gal/declare-aligner`

3.4.2 Case study: Dutch finantial institute

To demonstrate the practical utility of DECLAREALIGNER, a concrete example from the loan application process of the Dutch financial institute, i.e., dataset D1, is considered. The complete process model contains 16 constraints that govern the behavior of the process. For simplicity, this case study focuses on three relevant constraints:

- `NotCoExistence`(*A_ACCEPTED*, *A_DECLINED*): An application cannot be both accepted and declined at the same time.
- `NotSuccession`(*O_SELECTED*, *O_CREATED*): An offer cannot be created at any point after it has been selected.
- `Succession`(*O_CREATED*, *O_SENT*): After creating an offer, it must eventually be sent, and, if an order was sent, it must be eventually preceded by the creation of the order.

Table 3.5: Optimal alignment of the case study.

Log:	A_ACCEPTED	O_SELECTED	O_CREATED	O_SENT	O_SELECTED	O_CREATED	O_SENT	O_DECLINED
Model:	≫	≫	O_CREATED	O_SENT	≫	O_CREATED	O_SENT	O_DECLINED

Since the recorded traces are too long to reproduce here, consider the following relevant subsequence of a trace extracted from the recorded log data: $\sigma = \langle$*A_ACCEPTED*, *O_SELECTED*, *O_CREATED*, *O_SENT*, *O_SELECTED*, *O_CREATED*, *O_SENT*, *A_DECLINED*$\rangle$. Running DECLAREALIGNER on the full model and trace detects several errors in the traceand returns the optimal alignment shown in Table 3.5. This alignment includes three asynchronous moves: log move on *A_ACCEPTED*, and two log moves on *O_SELECTED*. These errors indicate that the activities corresponding to these moves should not have been performed in the trace, as they are incompatible with the constraints of the model. The algorithm determines that the cheapest way to reach conformance with the model is to not perform them.

The optimal alignment chosen by DECLAREALIGNER resolves the contradiction between accepting and declining an application, which violates the "an application cannot be both accepted and declined at the same time" business rule. Additionally, the trace shows two instances of an offer being selected, created, and sent, which contradicts the constraint "an

offer cannot be created at any point after it has been selected". An alternative alignment would require a log move on the creation of the offer instead, but this would violate the "if an order was sent, it must be eventually preceded by the creation of the order" constraint, necessitating further repairs in the trace. DeclareAligner carefully considers all possible alignments to ensure that the returned solution is optimal.

The detected problems are common issues in this loan application process. Many applications are accepted and later declined after a lengthy review process, causing confusion and delays. Furthermore, offers are often selected before they are created, suggesting issues with internal procedures. These problems can have serious consequences, such as incorrect or unfair treatment of applicants, and can damage the reputation of the financial institute. By using DeclareAligner to analyze process execution and detect errors, businesses can improve the quality and consistency of their processes, providing better service to their customers.

3.4.3 Ablation study

This ablation study evaluates the impact of each component in DeclareAligner to determine how they individually contribute to its overall performance. Various metrics are collected for each trace-model pair, such as execution time (*Time*), expanded states (*ExpStates*), and the timeouts (*Timeouts*). The results of the ablation study are shown in Table 3.6.

Early Pruning (Section 3.3.5.1) results in a significant decrease of 34.8% in average execution times. Furthermore, it enables the computation of many more alignments, reducing the number of timeouts by 985.

This improvement is mainly due to the pruning of a large number of states that would have led to unsolvable constraints, reducing the number of expanded states by 85.7%. By avoiding unnecessary computations and reducing the search space, this optimization ultimately is capable of computing complex alignments that would not be possible otherwise.

Example 3.10: Early Pruning ablation study

To demonstrate the effectiveness of the optimizations, a specific example from dataset D2[a] is considered. This process model comprises 10 constraints, representing a diverse range of constraint templates. The trace itself contains 50 events, and its optimal alignment has a cost of 4. Without any optimizations enabled, the algorithm computes

the alignment in 1.44 seconds, expanding 530 states in the process. This provides a useful reference point for evaluating the benefits of each optimization, illustrating why each of the proposed optimizations is particularly effective for this concrete example.

Enabling only Early Pruning significantly reduces the search space, decreasing the number of expanded states to 46 and the search time to approximately 0.46 seconds. Although this optimization prunes dead-ends by an order of magnitude, its impact on execution time is limited to a factor of three due to the additional overhead of checking all constraints for violations and suggesting repairs at each state expansion. However, as demonstrated by the comprehensive evaluation, the benefits of Early Pruning outweigh this extra cost, leading to improved overall performance.

[a]The example consists of the process model defined in *synthetic/10_constraints/10Constraints.xml* and the trace labeled as *Synthetic trace no.* 33 from the log file *.../3_constraints_inverted/log-1-50.xes.gz*.

Table 3.6: Ablation study results. *Time* and *ExpStates* represent the average over the complete dataset. *Timeouts* is the number of trace-model pairs for which the time limit was reached. *Reduction* values indicate improvements relative to the baseline algorithm, expressed as percentages for *Time* and *ExpStates*, and absolute differences for *Timeouts*.

EP	CP	GF	Time (s)	Time reduction	ExpStates	ExpStates reduction	Timeouts	Timeout reduction
			105.8	—	3,568.6	—	2,674	—
✓			68.9	34.8%	510.4	85.7%	1,689	985
	✓		76.8	27.4%	2,664.7	25.3%	1,973	701
		✓	54.2	48.8%	984.7	72.4%	1,170	1,504
✓	✓		55.2	47.8%	764.8	78.6%	1,348	1,326
✓		✓	36.4	65.6%	187.5	94.7%	706	1,968
	✓	✓	17.3	83.7%	423.9	88.1%	359	2,315
✓	✓	✓	**7.6**	**92.8%**	**107.0**	**97.0%**	**113**	**2,561**

Constraint Preprocessing (Section 3.3.5.2) yields a significant decrease in execution time (27.4%). This improvement is accompanied by a considerable reduction in the number of timeouts, with 701 fewer cases timing out.

The primary reason for this improvement is that constraint preprocessing allows the algorithm to start from a more advanced initial state, thereby avoiding the generation of numerous unnecessary states (25.3%). By performing shortcuts for highly likely operations upfront, this optimization reduces the overall search space, making it possible to compute alignments more efficiently.

Example 3.11: Constraint Preprocessing ablation study

Continuing the running example, enabling only Constraint Preprocessing yields a significant reduction in search space, decreasing the number of expanded states to 15 and the search time to approximately 0.09 seconds. This optimization is particularly effective in this case due to the presence of multiple chain constraints in the process model: `ChainResponse`(*act. 14*, *act. 15*), `ChainPrecendence`(*act. 16*, *act. 17*), and `NotChainSuccession`(*act. 22*, *act. 23*). By merging activities that appear consecutively in the trace and satisfy these constraints into a single LTGraph node during preprocessing, the algorithm avoids suggesting multiple repair actions during the search process, thereby preventing unnecessary widening of the search space. The fact that each of these constraints is activated multiple times in the example trace further amplifies the benefits of this optimization, making it a key factor in achieving the observed performance gains.

Grouped Fixes (Section 3.3.5.3) leads to a substantial reduction in execution time (48.8%) and a decrease in timeouts by 1,504 cases. This improvement is primarily due to the fact that grouped fixes enable the algorithm to prune the search space more effectively, reducing the number of expanded states by 72.4% and avoiding unnecessary exploration of intermediate states and their neighbors.

Example 3.12: Grouped Fixes ablation study

Continuing the running example, enabling only Grouped Fixes yields a significant reduction in search space, decreasing the number of expanded states to 123 and the search time to approximately 0.53 seconds. A notable aspect of this example is the presence of a `Response`(*act. 11*, *act. 12*) constraint, which is initially violated six times in the given trace. Fortunately, each repair can be performed in a single, consolidated action that simultaneously inserts a node and adds an arc to enforce the

desired behavior, effectively positioning the added activity in the correct region of the trace. By grouping these fixes together, the algorithm avoids generating intermediate states for this constraint, as well as several others in the example, resulting in a more efficient search process and improved overall performance.

Pairing any two optimizations generally leads to positive outcomes. This can be attributed to the fact that each optimization targets a distinct aspect of the search space, thereby reducing the number of expanded states required. Constraint Preprocessing focuses on moving the initial state closer to the goal, whereas Grouped Fixes avoid intermediate states when expanding neighbors. Meanwhile, Early Pruning eliminates dead-ends in the search space, preventing unnecessary exploration.

Combining all three optimizations yields the most substantial improvements across all tested metrics. The average execution time decreases by 92.8%, the expanded states are reduced by 97.0%, and timeouts are reduced by $2{,}561$ cases. By integrating these three optimizations, DeclareAligner is able to leverage their complementary strengths, resulting in a more efficient search strategy.

Example 3.13: All optimizations ablation study

To conclude the running example, enabling all three optimizations yields the most impressive performance gains, with a mere 5 expanded states and a search time of approximately 0.03 seconds. This shows how the optimizations are complementary and can be combined to achieve significant performance improvements.

3.4.4 State-of-the-art comparison

The remainder of the evaluation section assesses the performance of the DeclareAligner algorithm in comparison to the state-of-the-art methods. The state-of-the-art techniques for aligning event logs with declarative process models are DeclareReplayer [44] and PlannerBA/PlannerFD [54]. Both methods rely on an initial transformation of the declare constraints into automata, which are then utilized to guide the search algorithm. This sequential nature of automata execution leads to a search space that is built by incrementally appending alignment moves from start to finish. A key difference between the two approaches lies in their implementation. [44] employs the A* algorithm directly, whereas [54] transforms the generated automata into

classical planning problems, leveraging the capabilities of planning technology to solve the alignment problem. This distinction gives [54] an edge in certain scenarios, as the planning-based approach can efficiently handle complex search spaces. To provide a comprehensive understanding, this evaluation is supplemented by additional baselines:

- All proper subsets of optimizations described in Section 3.3.5 are considered, enabling only the optimizations in each subset.
- A naive algorithm that explores all possible alignments to find the optimal one.

To gain a deeper understanding of the interplay between optimization techniques, the DeclareAligner algorithm is evaluated under various configurations, each characterized by a unique combination of enabled optimizations. These are:

- B-None: no optimizations from Section 3.3.5 are applied.
- B-EP: only Early Pruning (EP) is enabled.
- B-CP: only Constraint Preprocessing (CP) is enabled.
- B-GF: only Grouped Fixes (GF) is enabled.
- B-EP-GF: both Early Pruning (EP) and Grouped Fixes (GF) are enabled.
- B-EP-CP: both Early Pruning (EP) and Constraint Preprocessing (CP) are enabled.
- B-CP-GF: both Constraint Preprocessing (CP) and Grouped Fixes (GF) are enabled.

Note that enabling all optimizations simultaneously results in the full algorithm, labeled as DeclareAligner. By systematically exploring these different configurations, this evaluation aims to elucidate the contribution of each optimization technique to the overall efficiency and effectiveness of the DeclareAligner algorithm, while also showcasing its improvements in the context of state-of-the-art performance.

A naive algorithm was implemented as another baseline that solves the optimal alignment problem for DECLARE process models and event logs. It leverages the A* search algorithm, starting from an initial state that represents an empty alignment, where no moves have been made. From this state, it generates neighboring states by considering all possible moves:

- If there are remaining events in the trace, where *act* is the next activity:
  - Generate a neighboring state by appending a synchronous move on *act*.
  - Generate a neighboring state by appending a log move on *act*.

- For each activity *act* present in the model, generate a neighboring state by appending a model move on *act*.

A state within this search space is considered a goal state if two conditions are met: there are no remaining events in the trace, and the model accepts the given trace. This acceptance check relies on a simple Declare model checker, which determines whether a trace perfectly conforms to the process model. The A* search algorithm guides the selection and expansion of neighboring states, albeit without utilizing a heuristic (i.e., the heuristic function always returns 0). This procedure repeats until a goal state is reached, at which point the optimal alignment is directly represented by this state. However, due to its simplicity, this naive approach generates an excessively large search space, needing substantial resources for exploration and rendering it impractical for realistic alignment problems.

Table 3.7 summarizes the results of the state-of-the-art approaches and DeclareAligner divided by each of the used dataset sources, and the aggregated total for the complete dataset. The results unequivocally demonstrate the superiority of DeclareAligner over existing state-of-the-art techniques, as it emerges as the top performer across all evaluated metrics. Notably, DeclareAligner achieves a significant lead in all categories, with the sole exception of the number of timeouts in the D2 dataset, where it narrowly trails behind the leading algorithm. However, even in this instance, DeclareAligner delivers substantial reductions in average execution time, underscoring its efficiency and scalability. Further analysis shows that most of the cases where the execution time of DeclareAligner is close to the state of the art are the simplest alignment problems in the dataset. This can be attributed to the initialization phase required by some proposed optimizations, which can introduce a delay for trivial alignments but prove highly beneficial for complex instances.

As anticipated, the performance of the naive algorithm is notably subpar, with execution times substantially exceeding those of DeclareAligner and other state-of-the-art algorithms. Specifically, it reaches the 5-minute time limit for 6,974 trace-model pairs. In contrast, the ablated versions of DeclareAligner manage even to surpass the state-of-the-art algorithms on some datasets and optimization combinations.

Table 3.7: Comparison of the average execution times and number of timeouts reached for each tested algorithm on the evaluated dataset, presented split by data source and as an aggregated total.

Dataset:	D1		D2		D3		All	
Algorithm	Time (s)	Timeouts	Time (s)	Timeouts	Time (s)	Timeouts	Time (s)	Timeouts
Naive	56.0	345	281.9	3,556	273.9	3,073	260.0	6,974
PlannerFD	6.3	411	117.4	832	124.2	703	111.4	1,946
DeclareReplayer	0.2	**0**	102.1	804	55.0	384	72.8	1,188
PlannerBA	8.5	355	34.3	**0**	65.6	45	46.2	400
B-None	1.4	117	182.2	1,839	44.1	718	105.8	2,674
B-EP	**0.1**	**0**	131.9	1,403	15.0	286	68.9	1,689
B-CP	1.4	121	120.8	1,223	44.0	629	76.8	1,973
B-GF	0.2	6	104.4	876	11.0	288	54.2	1,170
B-EP-GF	**0.1**	2	73.9	566	**3.3**	138	36.4	706
B-EP-CP	**0.1**	37	102.9	1,011	15.0	300	55.2	1,348
B-CP-GF	0.2	6	26.2	196	11.0	157	17.3	359
DeclareAligner	**0.1**	**0**	**13.0**	88	**3.3**	**25**	**7.6**	**113**

The results reveal that Early Pruning reduces average execution times from 105.8 to 68.9 seconds. This optimization also leads to a reduction in timeouts from 2,674 to 1,689, showcasing the effectiveness in reducing the search space that this optimization achieves. Constraint Preprocessing also yields significant improvements by allowing the algorithm to start from a more advanced initial state: it reduces the execution times from 105.8 to 76.8 seconds and the number of timeouts from 2,674 to 1,973. Lastly, Grouped Fixes avoids unnecessary exploration of intermediate states and their neighbors, reducing the execution times from 105.8 to 54.2 seconds and the number of timeouts from 2,674 to 1,170. Enabling any two optimizations at the same time contributes to improving the performance even further. Moreover, when all optimizations are combined (DeclareAligner), the algorithm achieves performance that is superior to its ablated counterparts across all metrics, with an average time of just 7.6 seconds and the number of timeouts to just 113. This suggests that the synergistic effect of combining the proposed optimizations yields a more efficient and effective alignment

Table 3.8: Ranking and statistical tests comparing execution times over the complete dataset.

(a) Average ranking of each tested algorithm.

Algorithm	Rank
DECLAREALIGNER	**1.24**
DeclareReplayer	2.73
PlannerBA	3.12
PlannerFD	3.55
Naive	4.37

(b) Results of the Friedman test and Holm post-hoc analysis, indicating overall differences between algorithms and specific pairwise differences.

Statistical test	p-value
DECLAREALIGNER vs all	$< 10^{-6}$
DECLAREALIGNER vs DeclareReplayer	$< 10^{-6}$
DECLAREALIGNER vs PlannerBA	$< 10^{-6}$
DECLAREALIGNER vs PlannerFD	$< 10^{-6}$
DECLAREALIGNER vs Naive	$< 10^{-6}$

algorithm, capable of outperforming existing state-of-the-art solutions in various scenarios.

To further substantiate the comparison, a ranking of all algorithms based on their execution times is computed. For each sample, the algorithms are ranked, with ties being assigned the average rank. These ranks are then averaged across all samples to obtain an overall ranking for each algorithm. To ensure that the results are statistically significant, the Friedman test is performed to assess overall differences between the algorithms. Subsequently, the Holm post-hoc test to identify specific pairwise differences. Notably, to avoid skewing the results, the variants of the proposed algorithm were excluded from the rankings and statistical tests. This ensures the focus remains on the main algorithm and prevents artificially high ranks that could misrepresent comparative performance.

The ranking and statistical tests are presented in Table 3.8. Prior to conducting these tests, it was verified that the necessary assumptions for their validity were met. Specifically, the ranking data consists of matched samples, as each algorithm's performance is evaluated on the same set of logs and models. Additionally, the samples are independent of one another, meaning that the performance of one algorithm on a given sample does not influence its performance on another sample.

It should also be noted that the Friedman test does not require normality of the data, which is beneficial in this case since execution times may not follow a normal distribution. Furthermore, the Friedman test does not assume variance homogeneity, making it suitable for these experiments even if the variances of the execution times differ across algorithms.

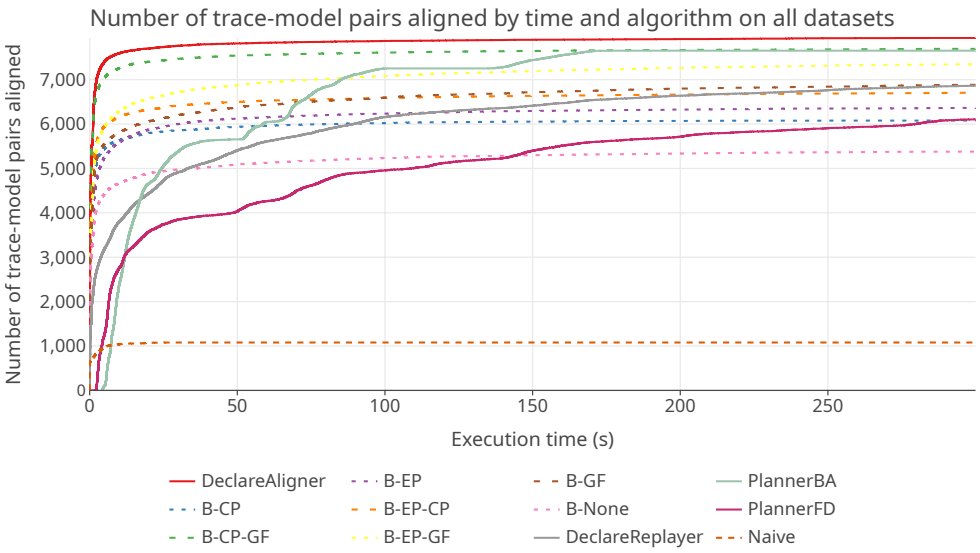


Figure 3.11: Number of trace-model pairs successfully aligned by each algorithm on all datasets.

As expected, the ranks show the clear advantage in execution times of DeclareAligner. The p-values indicate that the differences in performance among the algorithms are statistically significant, demonstrating that the proposed approach outperforms others in terms of computational efficiency. To quantify the magnitude of this effect, Kendall's W was calculated, a measure of the degree of agreement between the different execution times. The resulting effect size is 0.014. This is very low, which indicates that the performances of the algorithms do not agree with each other, suggesting that there are substantial differences between them. In fact, this lack of agreement is confirmed by the post-hoc test, which reveals significant pairwise differences between the performances of DeclareAligner and the state of the art.

To illustrate the efficiency and scalability of DeclareAligner compared to state-of-the-art algorithms, Figure 3.11 shows the number of trace-model pairs (y-axis) that can be solved within a given execution time (x-axis). Each algorithm is depicted as a distinct line, allowing for easy comparison of their performance across different time thresholds and number of logs solved.

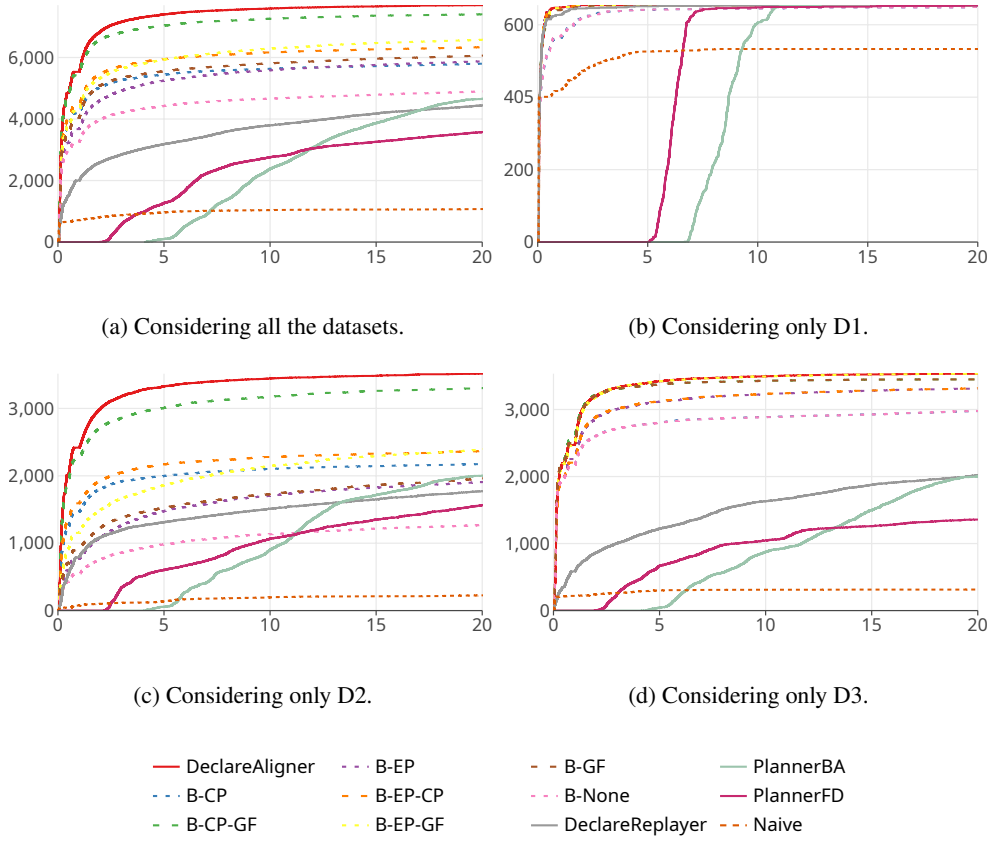
Figure 3.11 reveals that DeclareAligner significantly outperforms the state-of-the-art algorithms in terms of execution time. DeclareReplayer is also really fast to align the simplest

trace-model pairs, even being capable of outperforming DeclareAligner for some of the most straightforward alignments. However, its performance quickly degrades with complexity of the input problem, showing the superior scalability of DeclareAligner. At around 17 seconds, PlannerBA overtakes DeclareReplayer in terms of the number of trace-model pairs solved, taking second place. This occurs because both planner-based techniques require some initial time to convert the alignment task into a classic planning problem and the planner also needs to perform some preprocessing optimizations in order to solve harder problems faster. The sequential construction of the search space, inherent to both state-of-the-art approaches due to their reliance on automata, can lead to inefficiencies in exploring the vast possible search space of alignments. In contrast, the more direct approach to generating repair actions chosen by DeclareAligner yields improved performance, as illustrated in this figure.

The inclusion of ablated variants and the naive algorithm in the plots provides additional insights into the performance characteristics of DeclareAligner. Notably, the ablated variants exhibit reduced efficiency and scalability compared to DeclareAligner, highlighting the importance of the proposed optimizations. The naive algorithm, on the other hand, demonstrates poor performance across all datasets, aligning only 1,080 pairs when all datasets are considered, with no significant improvement over time. In contrast, DeclareAligner maintains its superior performance over all the alternatives, effectively leveraging its optimizations to achieve efficient alignments even in complex scenarios. These results underscore the value of the carefully designed optimizations in achieving scalable and efficient optimal alignments.

A notable achievement of DeclareAligner is its ability to align 90% of the tested trace-model pairs in 3.5 seconds or less per alignment, outperforming the next best state-of-the-art algorithm (PlannerBA) which requires up to 100 seconds per problem to reach the same number of solutions. Furthermore, the proposed optimizations enable DeclareAligner to successfully compute alignments for 231 trace-model pairs that remain unsolved by all other algorithms within a 5-minute time limit.

To offer a clearer and more detailed interpretation of the results, Figure 3.12 presents four supplementary subfigures that enhance the analysis provided in Figure 3.11. These plots focus specifically on a shorter time limit of 20 seconds, making performance differences between the tested algorithms more apparent. Each subfigure isolates a specific dataset source, as introduced earlier in this section. Figure 3.12(a) displays the aggregated results across all datasets, effectively serving as a zoomed-in version of Figure 3.11 and clearly highlighting the efficiency gains achieved by DeclareAligner.



(a) Considering all the datasets.

(b) Considering only D1.

(c) Considering only D2.

(d) Considering only D3.

(e) Shared legend for all subfigures.

Figure 3.12: Number of trace-model pairs successfully aligned by each algorithm in less than 20 seconds, also subdivided by dataset.

The remaining supplementary figures facilitate a detailed examination of performance differences within each dataset category. Figure 3.12(b) highlights DeclareAligner's efficiency in handling real-life data, demonstrating its strong capability to align process models effectively in practical scenarios. While the differences between DeclareAligner and DeclareReplayer are relatively small in this case, the planning-based techniques exhibit noticeable delays, primarily due to the overhead introduced during the problem translation

phase. Figure 3.12(c) showcases DeclareAligner's performance on synthetic data previously employed in state-of-the-art evaluations, providing a controlled setting to assess the algorithm's scalability and robustness. Figure 3.12(d) complements this by illustrating DeclareAligner's behavior on the extended synthetic dataset, which incorporates a wider variety of process model complexities through the use of additional constraint templates. In both cases, DeclareAligner demonstrates a clear advantage, successfully aligning nearly all trace-model pairs within the time limit of 20 seconds. In contrast, other state-of-the-art algorithms plateau significantly lower, typically aligning only about half of the available trace-model pairs, underscoring DeclareAligner's superior efficiency and reliability in more demanding scenarios.

3.5 Discussion

This chapter introduces DeclareAligner, an A*-based algorithm for efficiently computing optimal alignments of declarative process models. The contributions of this research are twofold. Firstly, a novel algorithmic approach has been proposed to leverage the inherent flexibility of declarative processes in calculating optimal alignments. Secondly, additional search optimizations and techniques have been developed and integrated to effectively handle complex scenarios.

The evaluation, conducted on 8,054 trace-model pairs from real-world and synthetic processes, demonstrates the substantial performance improvements of the proposed method over the state of the art. Notably, DeclareAligner successfully aligns 7,941 pairs within the 5 minutes time limit, outperforming PlannerBA (7,654), DeclareReplayer (6,866), and PlannerFD (6,108). Moreover, the proposed optimizations enable DeclareAligner to compute alignments for 231 pairs that are unsolvable by other algorithms within the same time frame. The results show that DeclareAligner achieves faster alignment times than any other algorithm for 7,000 of the evaluated trace-model pairs. The significance of this research lies in its ability to facilitate efficient conformance checking while preserving the accuracy of optimal alignments, ultimately leading to improved process quality and reduced costs through effective detection and diagnosis of log-related issues.

While the proposed algorithm has demonstrated its effectiveness, it still has some limitations, particularly with regards to scalability for large-scale models comprising hundreds of constraints, where the explosion of intermediate states resulting from numerous possible paths to optimal alignments poses a significant challenge. To fully realize the algorithm's potential, additional

research is necessary to investigate its performance on complex models and develop more strategies to enhance its efficiency. Furthermore, when applying the algorithm to real-world logs, it is important to consider the ethical implications, particularly with regard to sensitive data, and ensure that privacy and security are maintained through measures such as anonymizing log data or aggregating it to obscure sensitive information, thereby prioritizing data protection and upholding the highest standards of ethical responsibility.

In future research, the focus will shift towards adapting the proposed algorithm to accommodate data-aware declarative process models, ensuring retention of its high performance. Through extension to manage intricate data attributes and relationships, the goal is to offer a scalable conformance checking solution suitable for real-world applications demanding both precision and efficiency.

Chapter 4

Efficient Conformance Checking of Rich Data-Aware Declare Specifications

The contents of this chapter are partially extracted from the following publication:

Publication

Jacobo Casas-Ramos[a], Sarah Winkler[b], Alessandro Gianola[c], Marco Montali[b], Manuel Mucientes[a], and Manuel Lama[a]. Efficient conformance checking of rich data-aware declare specifications. In *Business Process Management*, pages 88–105, Cham, 2026. Springer Nature Switzerland.
DOI: 10.1007/978-3-032-02867-9_7

Conference ranking (ICORE, CORE2023): A

[a]CiTIUS, Universidade de Santiago de Compostela, Santiago de Compostela, Spain.
[b]Free University of Bozen-Bolzano, Bolzano, Italy.
[c]INESC-ID/Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal.

Chapters 2 and 3 introduced REACH and DeclareAligner respectively. They efficiently compute optimal alignments between event logs and procedural or declarative process models. Both methods focus on the control-flow aspect, furnishing a high-performance foundation for conformance checking within Process Mining.

This chapter extends the discussion to data-aware declarative models, specifically the Multi-Perspective DECLARE framework (MP-DECLARE) [51]. Declarative modeling is prized for its flexibility, which unfortunately expands the alignment search space and complicates conformance checking. Introducing event-level data further amplifies this complexity, as simultaneous consideration of control-flow and data perspectives becomes necessary. Despite these challenges, data-aware models yield richer, context-sensitive insights, thereby justifying the increased computational effort.

Increasingly, researchers are investigating alignment computation in multi-perspective process models that integrate both control-flow and data aspects [49, 28, 50]. This trend reflects the rising importance of data-aware models in process science, as well as the deeper understanding they afford compared to purely control-flow alignments [49]. However, devising alignment-based conformance checking for declarative, data-enriched specifications remains largely unresolved. Prior approaches concentrate on control-flow only, using representations such as Dynamic Condition Response (DCR) graphs [52], DECLARE constraints [53], or finite-trace Linear Temporal Logic (LTLf) [54]. To our knowledge, the sole venture into data-aware alignment within declarative settings is by Bergami et al. [28], which supports only numeric data and simple variable-to-constant comparisons —e.g., $x > 5$—, and omits inter-variable comparisons across events. The demand for more expressive multi-perspective constraints is well-documented [48, 49, 50].

It is unsurprising that earlier methods did not address rich constraints of this kind: computing optimal alignments is already computationally demanding, and data dependencies exacerbate the challenge. Moreover, previous methods using automaton-based techniques for capturing the language of a declarative specification [54, 28] are inadequate for data-rich domains. Indeed, extending beyond simple comparisons leads to undecidability issues [116].

To overcome these challenges, the current chapter presents Data-Aware DECLARE Aligner, an approach that reformulates the alignment problem a search over possible trace repairs. Two complementary strategies are integrated: an A* search mechanism inspired by the declarative alignment technique detailed in Chapter 3, and Satisfiability Modulo Theories (SMT) solving [55]. Data-Aware DECLARE Aligner defines a search space whose initial state encodes the original trace as an SMT formula. Each explored state undergoes SMT-based constraint violation detection, and any discrepancy triggers the generation of child states via targeted repairs. The algorithm proceeds until a goal state with minimal repair cost and no violations is reached, from which the optimal alignment can be reconstructed. The key contributions of this

chapter are:

1. A hybrid algorithm for computing optimal alignments, combining A* search with SMT-based reasoning to simultaneously handle control-flow and data conditions.
2. Full support for expressive data conditions, including quantitative temporal constraints and data correlations across activation and target events.
3. A suite of optimizations to enhance search efficiency and scalability:
    a) Leveraging SMT assumptions to avoid redundant solver work.
    b) Prioritizing violation repairs that generate fewer successor states.
    c) Detecting early and pruning dead-end states that cannot lead to a valid alignments.
    d) Eliminating unsatisfiable states through SMT-based consistency checks.

The correctness of the propposed method is proven and extensive experiments are conducted demonstrating its efficiency, even with intricate data conditions. This chapter presents an approach that not only surpasses the current state-of-the-art performance but also breaks new ground by supporting a wide range of rich data types and complex data expressions for the first time.

The rest of this chapter is structured as follows. Section 4.1 reviews related literature. In Section 4.2, the foundational definitions are established. The core algorithm is detailed in Section 4.3. Section 4.4 gathers some implementation details of interest. Empirical results and comparisons appear in Section 4.5. Finally, Section 4.6 offers concluding remarks and directions for future work.

4.1 Related Work

Although significant research has focused on computing optimal alignments for imperative process models that incorporate the data perspective [49, 50, 117, 39], there is a notable lack of work on data-aware alignment techniques for declarative models. Existing conformance checking methods for declarative models primarily focus on control-flow [44, 54, 46, 35, 52, 53]. Notably, they lack the capability to handle data conditions, a critical component of real-world

processes. Though some approaches have been proposed for conformance checking of data-aware DECLARE models, they have severe limitations. SQL queries have been employed to filter traces from a database that match the given specification, but this approach only considers exactly matching traces [118, 110]. Similarly, [119] uses constraint programming for trace analysis, but only supports global data. More comprehensive results are provided by the analysis framework [48], which reports activations, fulfillments, and violations of constraints. Moreover, importantly, none of the approaches [119, 110, 48] provides alignments, thus failing to offer detailed insights into the nature and extent of deviations between observed and expected behavior: This lack of nuanced analysis hinders the ability to identify root causes of non-conformance and implement targeted improvements, ultimately undermining the effectiveness of conformance checking efforts.

Closest to our work is the planning-based conformance checking approach [28], which does compute optimal alignments of data-aware DECLARE models. However, their treatment of the data perspective has severe limitations: data conditions can only refer to activation or target events, excluding correlation conditions that link the two. Moreover, data conditions are restricted to simple variable-to-constant comparisons, whereas our approach supports much more expressive data conditions over a wide range of data types including integers, bit-vectors, infinite-precision reals, and arrays. Specifically, we support the complete language of conditions specified in the SMT-LIB2 standard [55], requiring only that the underlying SMT theory is decidable. Overall, our approach offers a significant improvement over existing methods in terms of expressiveness and efficiency, making it a valuable tool for aligning complex processes while maintaining computational efficiency and expressive richness.

4.2 Preliminaries

In this section we introduce the required background about event logs, DECLARE with data, and alignments. We start with the data condition language and events.

Data conditions. We consider *sorts* $\Sigma = \{\texttt{bool}, \texttt{int}, \texttt{rat}, \texttt{string}\}$ for data payloads, with associated domains $\mathcal{D}(\texttt{bool}) = \mathbb{B}$, the booleans; $\mathcal{D}(\texttt{int}) = \mathbb{Z}$, the integers; $\mathcal{D}(\texttt{rat}) = \mathbb{Q}$, the rational numbers, and $\mathcal{D}(\texttt{string}) = \mathbb{S}$, finite strings. For a set of variables V and a sort $\sigma \in \Sigma$, V_σ denotes the subset of V of sort σ.

Definition 4.1: Data condition

A *data condition* over a set of variables V is an expression c according to the following grammar:

$$c := v_{\texttt{bool}} \mid b \mid n \geq n \mid r \geq r \mid r > r \mid s = s \mid c \wedge c \mid \neg c \qquad s := v_{\texttt{string}} \mid t$$
$$n := v_{\texttt{int}} \mid z \mid n + n \mid -n \qquad r := v_{\texttt{rat}} \mid q \mid r + r \mid -r$$

where $v_\sigma \in V_\sigma$ for $\sigma \in \Sigma$, $b \in \mathbb{B}$, $t \in \mathbb{S}$, $z \in \mathbb{Z}$, and $q \in \mathbb{Q}$. The set of data conditions over a set of variables V is denoted by $C(V)$.

We use data conditions as in Definition 4.1 in this chapter to have a concrete language to refer to, but our implementation actually allows for arbitrary conditions in the SMT-LIB2 language [55] that is supported by the SMT solver of choice. In the sequel, we assume that the underlying SMT theory is decidable, though; this restriction is required to provide correctness guarantees.

Event logs. Below, we assume an arbitrary infinite set Id of event identifiers; and a set $\mathcal{A}$ of activities, where elements $a \in \mathcal{A}$ are denoted by lower-case letters. We consider the following notions of data-aware events, traces, and event logs:

Definition 4.2: Event log with data

An *event* e is a triple $e = (\iota, a, \alpha)$ such that $\iota \in Id$, $a \in \mathcal{A}$ is an activity, and α is a partial assignment that maps variables in V to elements of their domain. Given a set of events E, a *trace* $\mathbf{e}$ is a finite sequence of events in E, that is, $\mathbf{e} \in E^*$; and an *event log* is a multiset of traces. The domain of an assignment α is denoted $dom(\alpha)$.

DECLARE. Table 4.1 lists the DECLARE templates used in this chapter[1]. We call a *DECLARE constraint* an expression that is obtained from a DECLARE template by substituting the upper-case template variables by activities in $\mathcal{A}$. Constraints based on templates (6)–(9) and (10)–(12) are called *response* and *precedence constraints*, respectively, while constraints using (13)–(15) are *negation constraints*.

DECLARE templates, as well as the derived constraints, have *activations* and *targets*. Intuitively, an activation is an event whose occurrence imposes the (non) occurrence of other

[1] Our implementation supports in fact the full list in Table 3.2.

Table 4.1: Most of the supported DECLARE templates.

(1)	$\mathrm{Existence}(n, A)$:	A occurs at least n times.
(2)	$\mathrm{Absence}(n, A)$:	A occurs at most $n-1$ times.
(3)	$\mathrm{Init}(A)$:	A is the first activity.
(4)	$\mathrm{End}(A)$:	A is the last activity.
(5)	$\mathrm{Choice}(A, B)$:	Either A or B, or both, occur.
(6)	$\mathrm{RespondedExistence}(A, B)$:	If A occurs, B also occurs.
(7)	$\mathrm{Response}(A, B)$:	If A occurs, B follows.
(8)	$\mathrm{AlternateResponse}(A, B)$:	If A occurs, B follows without an A in between.
(9)	$\mathrm{ChainResponse}(A, B)$:	If A occurs, B is the next activity.
(10)	$\mathrm{Precedence}(A, B)$:	If B occurs, A precedes it.
(11)	$\mathrm{AlternatePrecedence}(A, B)$:	If B occurs, A precedes it without a B in between.
(12)	$\mathrm{ChainPrecedence}(A, B)$:	If B occurs, A is the previous activity.
(13)	$\mathrm{NotResponse}(A, B)$:	If A occurs, B does follow.
(14)	$\mathrm{NotRespondedExistence}(A, B)$:	If A occurs, B does not.
(15)	$\mathrm{NotChainResponse}(A, B)$:	If A occurs, B is not the next activity.

events. These other events are called targets. For the templates in Table 4.1, in all response and negation templates, variable A is the activation and B the target; while in all precedence templates, B is the activation and A the target. In the remaining patterns, both A and B are targets. Given a DECLARE constraint φ, an activity is an activation (resp. target) activity in φ if it is substituted for an activation (resp. target) variable in the underlying template.

We consider *Multi-Perspective* DECLARE constraints that include data conditions. To that end, for the remainder of the chapter we fix a set of sorted *process variables* V. Intuitively, these variables are considered the payload of activities. They are maintained along the entire trace, but may change their values. For a set Set, let $V^{Set} = \{v_s \mid v \in V \text{ and } s \in Set\}$ be a set of labelled variables that contains a copy of each variable in V for each element in Set. In particular, for $a \in \mathcal{A}$, the idea is that v_a represents the value of v while observing activity a.

Definition 4.3: DECLARE constraint with data conditions

A *DECLARE constraint with data conditions* is a quadruple $\langle \varphi, c_{act}, c_{tgt}, c_{cor} \rangle$ consisting of a DECLARE constraint φ and data conditions c_{act}, c_{tgt} and c_{cor}. Precisely, for $a \in \mathcal{A}$ the activation and $T \subseteq \mathcal{A}$ the target activities of φ:

- $c_{act} \in C(V^{\{a\}})$ is called the *activation condition*,
- $c_{tgt} \in C(V^{T})$ is called the *target condition*, and
- $c_{cor} \in C(V^{\{a\} \cup T})$ is called the *correlation condition*.

Intuitively, c_{act} constrains the data variables while the activation activity is observed, c_{tgt} the data variables while the target activity is observed, and c_{cor} expresses relationships between the data variables of both activities. For DECLARE constraints φ without activation, we assume that all but c_{tgt} are $\top$. For simplicity of presentation, we assume that the activation and target activity are different, (though in our implementation this is not required). A *DECLARE specification* $\mathcal{M}$ is a set of DECLARE constraints with data. In the sequel, if no confusion can arise, we refer to DECLARE constraints with data simply by *constraints*.

Example 4.1: Running example

As a running example, we use the set of variables $V = \{x\}$, activities $\mathcal{A} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ and the specification $\mathcal{M}$ that consists of the following two constraints ψ_1 and ψ_2, where for readability we write $a.v$ instead of v_a, for $a \in \mathcal{A}$:

- $\psi_1 = \langle \texttt{ChainResponse}(\mathsf{a}, \mathsf{c}), \top, \top, \mathsf{c}.x > \mathsf{a}.x \rangle$: This specifies that each occurrence of a must be directly followed by an event with c such that the value of x associated with activity c is greater than the value of x seen with a.
- $\psi_2 = \langle \texttt{AlternatePrecedence}(\mathsf{c}, \mathsf{b}), \mathsf{b}.x \geq 0, \mathsf{c}.x \neq 0, \mathsf{c}.x < \mathsf{b}.x \rangle$: This states an alternate precedence relationship between the activation b and the target c, demanding that if the value of x seen with b is non-negative, an activity c must occur before activity b, without any other b activities with $x \geq 0$ in between. Furthermore, the x value of c must be lower than the x value of b.

The semantics of DECLARE constraints with data is the same as in [28]. For the sake of completeness, we provide a concise recapitulation of this definition in Appendix A. The set of all traces that satisfy all constraints in $\mathcal{M}$ is denoted by $runs(\mathcal{M})$. We assume for our approach

that $runs(\mathcal{M}) \neq \emptyset$.

Alignments. We aim to design a conformance-checking procedure that, given a trace and a DECLARE specification $\mathcal{M}$, finds an optimal alignment of $\mathbf{e}$ and a run of $\mathcal{M}$. Typically, when constructing alignments, not all events in the trace can be put in correspondence with an event in a run, and vice versa. Hence we use a "skip" symbol $\gg$ and consider the extended set of events $E^{\gg} = E \cup \{\gg\}$.

Definition 4.4: Legal move (Data-Aware DECLARE)

For a set E of events as above, a pair $(e, f) \in E^{\gg 2} \setminus \{(\gg, \gg)\}$ is called *legal move* iff one of the following cases applies: it is a

- *log move* if $e \in E$ and $f = \gg$;
- *model move* if $e = \gg$ and $f \in E$;
- *edit move* if $(e, f) \in E^2$, $(e, f) = ((\iota, a, \alpha), (\iota, a, \alpha'))$, $\mathcal{D}(\alpha) = \mathcal{D}(\alpha')$ and $\exists v \in \mathcal{D}(\alpha)$ such that $\alpha(v) \neq \alpha'(v)$;
- *synchronous move* if $(e, f) \in E^2$ and $e = f$.

We denote by $Moves$ the set of all moves.

For a sequence of moves $\gamma = \langle (e_1, f_1), \ldots, (e_n, f_n) \rangle$, the *log projection* $\gamma|_L$ of γ is the maximal subsequence $e'_1, \ldots, e'_i$ of $e_1, \ldots, e_n$ such that $e'_1, \ldots, e'_i \in E^*$, that is, it contains no $\gg$ symbols. Similarly, the *model projection* $\gamma|_M$ of γ is the maximal subsequence $f'_1, \ldots, f'_j$ of $f_1, \ldots, f_n$ such that $f'_1, \ldots, f'_j \in E^*$.

Definition 4.5: Alignment (Data-Aware DECLARE)

Given a DECLARE model $\mathcal{M}$, a sequence of moves γ is an *alignment* of a trace $\mathbf{e}$ against $\mathcal{M}$ if $\gamma|_L = \mathbf{e}$, and $\gamma|_M \in runs(\mathcal{M})$. The set of alignments for a trace $\mathbf{e}$ wrt. $\mathcal{M}$ is denoted by $Align(\mathcal{M}, \mathbf{e})$.

Example 4.2: Alignments for the running example

Consider the trace $\mathbf{e} = \langle (\#_1, \mathsf{a}, \{x = 0\}), (\#_2, \mathsf{b}, \{x = 2\}) \rangle$. The following are two

possible alignments for $\mathbf{e}$ against the model from Example 4.1:

$$\gamma_1 = \begin{array}{|c|c|c|c|c|} \hline \mathsf{a} & \{x=0\} & \gg & \mathsf{b} & \{x=2\} \\ \hline \mathsf{a} & \{x=0\} & \mathsf{c} \quad \{x=1\} & \mathsf{b} & \{x=2\} \\ \hline \end{array} \qquad \gamma_2 = \begin{array}{|c|c|c|c|c|} \hline \mathsf{a} & \{x=0\} & \gg & \mathsf{b} & \{x=2\} \\ \hline \mathsf{a} & \{x=0\} & \mathsf{c} \quad \{x=3\} & \gg & \\ \hline \end{array}$$

Each move (e, f) is shown in a column, including e in the first row and f in the second row. Since event identifiers are irrelevant in alignments, we omit them.

A *cost function* is a mapping $\kappa\colon \mathit{Moves} \to \mathbb{R}^+$ that assigns a cost to every move. It is naturally extended to alignments as follows.

Definition 4.6: Alignment cost and optimal alignment (Data-Aware DECLARE)

Given $\gamma \in \mathit{Align}(\mathcal{M}, \mathbf{e})$ as before, the *cost* of γ is defined as the sum of the costs of its moves, that is, $\kappa(\gamma) = \sum_{i=1}^{n} \kappa(e_i, f_i)$. Moreover, γ is *optimal for* $\mathbf{e}$ *and* $\mathcal{M}$ if $\kappa(\gamma)$ is minimal among all alignments for $\mathbf{e}$ and $\mathcal{M}$, namely there is no $\gamma' \in \mathit{Align}(\mathcal{M}, \mathbf{e})$ with $\kappa(\gamma') < \kappa(\gamma)$.

In this chapter, we will use the standard cost function κ that assigns $\kappa(e, f) = 1$ if (e, f) is a log or model move, $\kappa(e, f) = 0$ if (e, f) is a synchronous move, and for an edit move $(e, f) = ((\iota, a, \alpha), (\iota, a, \alpha'))$, $\kappa(e, f) = |\{\alpha(v) \neq \alpha'(v) \mid v \in V\}|$.

4.3 Data-Aware DECLARE Aligner

In this section, we outline the conceptual approach of the Data-Aware DECLARE Aligner. Given a DECLARE specification $\mathcal{M}$ and a trace $\mathbf{e}$, the aim is to find an optimal alignment of $\mathbf{e}$ wrt. $\mathcal{M}$. To that end, the basic idea is to start with the event sequence in $\mathbf{e}$, and subsequently *repair* it until an event sequence is obtained that satisfies all constraints in $\mathcal{M}$. To navigate through a large search space of possible repairs and respective alignments while ensuring an optimal solution, our approach leverages the A* algorithm.

We use the term *state* to refer to a representation of a candidate alignment. The formal definition of a state is given below; intuitively, each state consists of a partially ordered set of events together with data conditions, effectively acting as a candidate alignment which need not yet satisfy all constraints. Moreover, each state has a *cost*, reflecting the cost of alignments extracted from it.

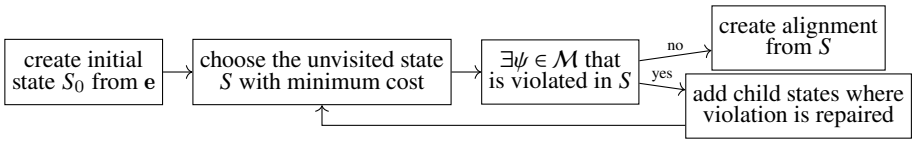


Figure 4.1: Overview of the approach.

An overview of the approach is sketched in Figure 4.1: the initial state S_0 represents the set of events in the input trace, ordered as in $\mathbf{e}$, and with data conditions that reflect the variable assignments. The procedure then selects the previously unvisited state S of minimal cost. It is checked whether there are remaining constraint violations in S. In this case, all possible *repairs* are applied to S creating a new child state from each repair, and another search iteration is performed. Otherwise, an optimal alignment for $\mathbf{e}$ is reconstructed from S.

4.3.1 State definition

As mentioned above, a state contains a partially ordered set of events, and data conditions on their payloads. In order to express conditions that involve variables in all events, we need, as a technicality, labelled variables: For an event $e = (\iota, a, \alpha)$, let $V^e = \{v_\iota \mid v \in V\}$ be a copy of the set V where each variable is labelled by the id of e. For a set of events E, let $V^E = \bigcup_{e \in E} V^e$ be the set of variables for all events in E. A state with set of events E can then use data conditions (cf. Definition 4.1) on V^E to refer to the events' payloads. In the sequel we also assume that V contains a special variable of type integer, and ι will denote the timestamp of event with id ι. To reason about partial orderings of events in E, states use *ordering conditions*, defined next:

Definition 4.7: Ordering condition

An *ordering condition* o for a set of events E is of the following form, where $e, e' \in E$, $a \in \mathcal{A}$ is an activity, and c is a data condition as in Definition 4.1:

$$o := e < e' \mid e \ll e' \mid \mathit{first}(e) \mid \mathit{last}(e) \mid e <^a_{[c]} e' \mid \neg o \mid \top$$

Here $e < e'$ expresses that e happens before e', $e \ll e'$ that e happens before e' without any other event in between, $\mathit{first}(e)$ that e is the first, $\mathit{last}(e)$ that e is the last element, and $\top$ is

a condition that is always true. Somewhat more complex, $e <^{a}_{[c]} e'$ expresses that e happens before e' without an event e'' in between that has activity a and satisfies c, where c is supposed to be a data condition over $V^{e''}$. A set of ordering conditions on E in *satisfiable* if there exists a topological sort of E that satisfies all conditions. A set of ordering conditions O is said to *entail* an ordering condition q, denoted $O \models q$, if $\bigwedge O \wedge \neg q$ is unsatisfiable. Note that the ordering conditions are defined to closely align with the semantics of the supported DECLARE templates, as clarified in Def. 4.9.

Definition 4.8: State (Data-Aware DECLARE Aligner)

A *state* is a pair $S = \langle E, C \rangle$ where E is a set of events, and C is a set of ordering conditions on E and data conditions over V^E.

A state $\langle E, C \rangle$ thus represents a set of events E that is partially ordered by the ordering conditions in C, and where payloads of events are constrained by the data conditions in C.

For a trace $\mathbf{e} = \langle e_1, \ldots, e_n \rangle$, let $E(\mathbf{e}) = \{e_1, \ldots, e_n\}$ be its set of events, $O(\mathbf{e}) = \{e_i < e_{i+1} \mid 1 \leq i < n\}$ be the set of ordering conditions that capture the event ordering in $\mathbf{e}$, and $D(\mathbf{e})$ the conjunction of all equations $v_\iota = \alpha(v)$ such that an event $e = (\iota, a, \alpha)$ occurs in $\mathbf{e}$ and $v \in dom(\alpha)$. The *initial state* is $\langle E(\mathbf{e}), O(\mathbf{e}) \cup D(\mathbf{e}) \rangle$, it serves as the starting point for the exploration of the search space. Note that we use formulas that mix ordering and data conditions; we will explain in the next section how standard SMT solvers can be used to perform satisfiability checks of such formulas.

Example 4.3: Search space for the running example

Consider the trace $\mathbf{e}$ in Example 4.2 with events $e_1 = (\#_1, \mathsf{a}, \{x = 0\})$ and $e_2 = (\#_2, \mathsf{b}, \{x = 2\})$. If no confusion can arise, we write $\mathsf{a}.x$ rather than $x_{\#_1}$ etc. for readability. The initial state is $S_0 = \langle \{e_1, e_2\}, (e_1 < e_2) \wedge (\mathsf{a}.x = 0) \wedge (\mathsf{b}.x = 2) \rangle$. Here $e_1 < e_2$ expresses that e_1 happens before e_2; the remaining conditions fix the values of x in the two events. Figure 4.2 shows most of the search space for $\mathbf{e}$ and the specification from Example 4.1 —the complete search space is shown in Figure B.1. States are shown as boxes, S_0 being the box on top. The events of a state are shown as boxes with activities within the state, and arrows in between them indicate ordering conditions. Here $e_1 < e_2$ is displayed by an arrow $e_1 \rightarrow e_2$, $e_1 \ll e_2$ by $e_1 \twoheadrightarrow e_2$, and the condition

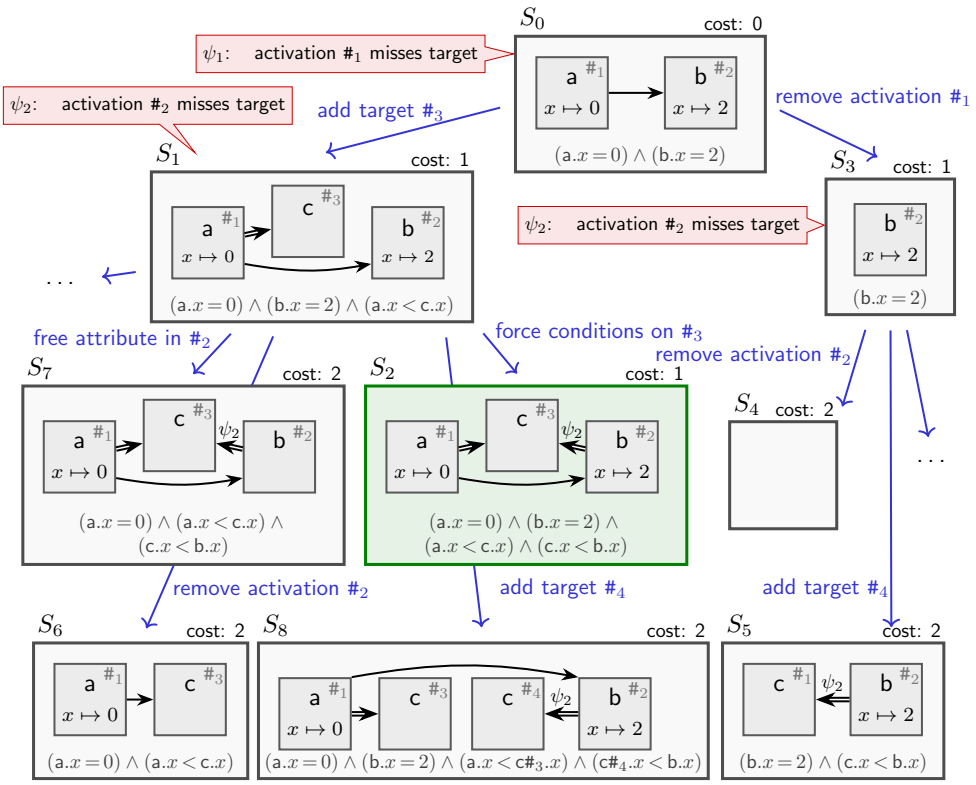


Figure 4.2: Search space for the running example.

$e_1 <^{\mathsf{b}}_{[\mathsf{c}.x<\mathsf{b}.x]} e_2$ obtained from ψ_2 by $e_1 \overset{\psi_2}{\Longrightarrow} e_2$. The formulas at the bottom of states specify data conditions. The states S_1–S_8 are obtained from S_0 by applying repairs; we will explain below how this is done.

4.3.2 Constraint violations

We next define when constraints are violated in a state. To that end, we need some additional notation: given a DECLARE constraint ψ and events $e, e' \in E$, we denote by $Ord(\psi, e, e')$ the ordering conditions imposed by ψ between an activation event e and a target event e', defined as follows:

Definition 4.9: Constraint violation

Let e, e' be events and $\psi = (\varphi, c_{act}, c_{tgt}, c_{cor})$ a constraint. For templates φ with an activation, we define $Ord(\psi, e, e')$ as $e < e'$ (resp. $e' < e$) if φ is based on a `Response` (resp. `Precedence`) template, $e \ll e'$ (resp. $e' \ll e$) if it is a `ChainResponse` (resp. `ChainPrecedence`) template, $\neg(e < e')$ for `NotResponse`, $\neg(e \ll e')$ for `NotChainResponse`, $e <^{a}_{[c_{act}]} e'$ for `AlternateResponse`, and $e' <^{a}_{[c_{act}]} e$ for `AlternatePrecedence`. In the last cases, a is the activation activity of φ. For constraints φ without activation, let $Ord(\psi, e)$ be $first(e)$ or $last(e)$ if φ is an `Init` or `Last` constraint, respectively. In all other cases, $Ord(\psi, e) = \top$.

We also need to instantiate data conditions for events. To that end, given a DECLARE constraint $\psi = (\varphi, c_{act}, c_{tgt}, c_{cor})$ and an event $e = (\iota, a, \alpha)$ such that a is an activation activity for φ and b a target activity, we denote by $[c_{act}](e)$ (resp. $[c_{tgt}](e)$) the condition obtained from c_{act} (resp. c_{tgt}) by substituting v_a (resp. v_b) with v_ι for each $v \in V$. Similarly, for another event $e' = (\delta, b, \alpha')$, $[c_{tgt} \wedge c_{cor}](e, e')$ denotes the condition obtained from $c_{tgt} \wedge c_{cor}$ by substituting variables v_a by v_ι, and v_b by v_δ for all $v \in V$.

The first kind of violation is a *missing target*; intuitively, it applies if a constraint ψ can be activated but might lack a target that satisfies all conditions.

Definition 4.10: Missing target

A constraint $(\varphi, c_{act}, c_{tgt}, c_{cor})$ has a *missing target* violation in state (E, C) if one of the following cases applies:

- φ is a response or precedence constraint with activation activity a and there is an $e = (\iota, a, \alpha) \in E$ such that $C \wedge [c_{act}](e)$ is satisfiable, but no $e' \in E$ with target activity such that $\bigwedge C \wedge [c_{act}](e) \models Ord(\varphi, e, e') \wedge [c_{tgt} \wedge c_{cor}](e, e')$; or
- φ is of the form `Existence`(n, a), and $e_1, \ldots, e_k$ are all events with activity a in E but $\bigwedge C \models \Sigma_{i=1}^{k} ite([c_{tgt}](e_i), 1, 0) \geq n$ does not hold; or
- φ is an `Init`, `End`, or `Choice` constraint and $e_1, \ldots, e_k$ are all events with target activity in E but $\bigwedge C \models \bigvee_{i=1}^{k} Ord(\psi, e_i) \wedge [c_{tgt}](e_i)$ does not hold.

Here $ite(b, d_1, d_2)$ abbreviates an if-then-else expression. In the first case of Definition 4.10,

e is called *activation event*.

Figure 4.2 shows three cases of missing target violations for the constraints in Example 4.1: in state S_0, ψ_1 is activated by the event $\#_1$ with activity a, but no target event with activity c is present. In S_1 and S_3, ψ_2 is violated: in S_3 since no event with activity c occurs, and in S_1 because, even though an event with activity c occurs, namely $\#_3$, its conditions do not entail $\mathsf{c}.x < \mathsf{b}.x$ and $\#_3 <^{a}_{[c_{act}]} \#_2$.

The second kind of violation is dual in that it signals too many targets.

Definition 4.11: Excessive target

A constraint $\psi = (\varphi, c_{act}, c_{tgt}, c_{cor})$ has an *excessive target* violation in a state $S = (E, C)$ if one of the following cases applies:

- φ is of the form $\texttt{Absence}(n, a)$ and there are n events $e_1, \ldots, e_n$ in E with activity a such that $\bigwedge C \wedge \bigwedge_{i=1}^{n} [c_{tgt}](e_i)$ is satisfiable;
- φ is a negation constraint, some $e_0, e_1 \in E$ have activation and target activity, resp., and $\bigwedge C \wedge Ord(\psi, e_0, e_1) \wedge [c_{act}](e_0) \wedge [c_{tgt} \wedge c_{cor}](e_0, e_1)$ is satisfiable.

The events $e_1, \ldots, e_n$ in Definition 4.11 are called *excessive target events*.

For instance, for the states in Figure 4.2, a constraint $\langle \texttt{NotResponse}(\mathsf{a}, \mathsf{b}), \mathsf{a}.x \geq 0, \top, \mathsf{b}.x > \mathsf{a}.x \rangle$ would have an excessive target violation in states S_0 and S_1, but not in S_3. On the other hand, $\langle \texttt{Absence}(\mathsf{b}), \top, \mathsf{b}.x = 3 \rangle$ would be violated in state S_7, but not in S_0 where the data conditions exclude $\mathsf{b}.x = 3$.

A constraint ψ is *violated* in a state if it has a missing or excessive target. A state $S = \langle E, C \rangle$ is a *goal state* if no constraint in $\mathcal{M}$ is violated in S and C is satisfiable. In Figure 4.2, all leaves of the search tree (S_2 and S_4–S_8) are goal states.

4.3.3 Repairing violations

Our approach subsequently expands the search space by selecting a state where a constraint is violated, generating *child states* by repairing the violation in different ways. Four kinds of repairs are distinguished:

- addition of an event, which will be reflected as a model move in the alignment;

- removal of an event that stems from the trace, corresponding to a log move in the alignment;
- freeing a data attribute in an event that stems from the trace, corresponding to an edit move; and
- enforcement of conditions.

The applicable repairs and resulting states depend on the violated constraint $\psi = (\varphi, c_{act}, c_{tgt}, c_{cor}) \in \mathcal{M}$ and current state $S = \langle E, C \rangle$. First, if ψ has an activation event $e_{act} \in E$, the following repairs are applied for both missing and excessive target violations to *disable* the activation:

(1) *Removing an activation event.* This repair only applies if e_{act} stems from the trace $\mathbf{e}$. The resulting state is $S' = \langle E \setminus \{e_{act}\}, C' \rangle$ where C' is like C with conditions involving e_{act} removed.

(2) *Freeing a data attribute.* This applies to an event $e_{act} = (\iota, a, \alpha)$ from the trace $\mathbf{e}$ if α does not satisfy $\neg[c_{act}](e_{act})$. The repair removes an assignment $\alpha(v)$ of e_{act} for some $v \in V$. For $C' = C \setminus \{v_\iota = \alpha(v)\} \cup \{\neg[c_{act}](e_{act})\}$, the new state is $S' = \langle E, C' \rangle$.

(3) *Enforcing the negated activation condition.* The resulting state is $S' = \langle E, C' \rangle$ with $C' = C \cup \{\neg[c_{act}](e_{act})\}$.

If the violation is a missing target, then a target event can be added, or an existing event with the correct activity can be enforced to satisfy the data conditions, or in some cases events can be removed that block ordering conditions. More precisely, the following repairs apply:

(4) *Adding a target event.* A new state is of the form $S' = \langle E \cup \{e\}, C' \rangle$ where $e = (\iota, a, \emptyset)$ is a new event with fresh identifier ι. If ψ has an activation and e' is the activation event, then $C' = C \cup \{Ord(\psi, e', e), [c_{act}](e'), [c_{tgt} \wedge c_{cor}](e', e)\}$. Otherwise, $C' = C \cup \{Ord(\psi, e), [c_{tgt}](e)\}$.

(5) *Freeing a data attribute.* This applies to events $e = (\iota, a, \alpha)$ in E that stem from the trace $\mathbf{e}$ and have the target activity but do not satisfy $[c_{tgt} \wedge c_{cor}](e', e)$ if ψ has an activation event e', resp. $[c_{tgt}](e)$ otherwise. The repair removes some assignment $\alpha(v)$ for $v \in V$, which can avoid the violation. The new state is $S' = \langle E', C \setminus \{v_\iota = \alpha(v)\} \rangle$ where E' is like E but where e is modifed to (ι,a,α') such that α' is like α except being undefined for v.

(6) *Enforcing conditions.* This applies to an event $e \in E$ with activity a, i.e., a potential target event. The new state is $S = \langle E, C' \rangle$, where if ψ has an activation, and e' is the activation event that caused the missing target, then $C' = C \cup \{Ord(\psi, e', e)\} \cup \{[c_{act}](e'), [c_{tgt} \wedge c_{cor}](e', e)\}$; otherwise, $C' = C \cup \{Ord(\psi, e), [c_{tgt}](e)\}$.

(7) *Removing a blocking event.* This applies if an event $e_t \in E$ with activity a is according to the ordering conditions in C not in the right position to act as target for ψ, but removing another event e from the trace can make room for e_t. E.g., `Init` constraints delete the first event, and `ChainResponse` constraints remove the events directly succeeding the activation. The resulting state is $S' = \langle E \setminus \{e\}, C' \rangle$ where C' is like C with conditions on e removed.

If ψ has an excessive target, we can either remove an excessive target event, or change the data conditions such that an event with target activity no longer acts as a target. Precisely, the following fixes apply:

(8) *Removing excessive target events.* This works like (1) above, but removes excessive target events if they stem from the trace.

(9) *Freeing a data attribute.* This repair is similar to (2) above, but it applies if there is an excessive target event $e = (\iota, a, \alpha)$ in E that stems from the input trace. However, we now enforce the negation of target and correlation conditions. The resulting state is $S' = \langle E', C' \rangle$ where E' is like E but where e is modified to (ι, a, α') such that α' is like α except that it is undefined for $v \in V$. Let $\widehat{C} = C \setminus \{v_\iota = \alpha(v)\}$. If there is an activation event e', we set $C' = \widehat{C} \cup \{\neg[c_{tgt} \wedge c_{cor}](e', e)\}$; otherwise, $C' = \widehat{C} \cup \{\neg[c_{tgt}](e)\}$.

(10) *Enforcing negated conditions.* Let $e \in E$ be an excessive target event. There are two resulting states $S' = \langle E, C' \rangle$ and $S'' = \langle E, C'' \rangle$. If there is an activation event e', then $C' = C \cup \{\neg[c_{tgt} \wedge c_{cor}](e', e)\}$; otherwise, $C' = C \cup \{\neg[c_{tgt}](e)\}$. Moreover, $C'' = C \cup \{\neg Ord(e', e)\}$.

Note that *all* applicable repairs are applied in all possible ways. For instance, when freeing a data attribute, a new state is generated for every event e and every variable assignment in e that satisfies the conditions in (2). Also, if there is a missing target violation and φ is a `Choice` constraint having two targets, a child state is created for each possible target.

Example 4.4: Application of repairs for the running example

For example, in Figure 4.2, S_1 is obtained from S_0 by adding a target event $\#_3$ (repair (4)); S_3 is obtained from S_0 by removing the activation event $\#_1$ (repair (1)); S_7 is obtained from S_1 by freeing the data attribute x in event $\#_2$ (repair (2)); and S_2 is obtained from S_1 by forcing conditions on event $\#_3$ (repair (6)).

4.3.4 A*-based search

Starting from the initial state that represents the input trace $\mathbf{e}$, our algorithm subsequently chooses a state with a violation and generates child states by applying all possible repairs. By a *search space* for $\mathbf{e}$ and $\mathcal{M}$, we mean below a graph of states where the root is S_0, and all states have as children the states obtained by all possible repairs, if any. To guide the search, the A* algorithm maintains for each state S a cost $cost(S) \in \mathbb{R}$, which can be shown to match exactly the cost of alignments extracted from S. The initial state has cost 0. When expanding a state S, the cost of a child state S' is determined by the applied repair: when adding or removing events, or freeing a data attribute, we have $cost(S') = cost(S) + 1$; when forcing condition satisfaction, $cost(S') = cost(S)$. In Figure 4.2, each state is labelled with its respective cost.

As repairs cause an increase in cost, by a fair exploration of the search space, the A* algorithm can conclude at some point that all goal states possibly detected in the future will have a higher or equal cost than the goal states found so far. At this point the search terminates, returning a goal state with minimal cost.

4.3.5 Alignment extraction

From a goal state $S = \langle E, C \rangle$, we extract an alignment as described by the pseudocode in Alg. 4.1. The first step is to obtain an SMT model μ of the conditions $\bigwedge C$. This induces a list of model events $\mathbf{f} = \langle f_0, \ldots, f_{m-1} \rangle$ that satisfies all ordering conditions, and where for each $f_j = (\iota_j, a_j, \alpha_j)$ the assignment α_j is given by $\alpha_j(v) = \mu(v_{\iota_j})$ for all $0 \leq j < m$.

Alg. 4.1 then walks simultaneously along $\mathbf{e}$ and $\mathbf{f}$, using i as an index for $\mathbf{e}$ and j for $\mathbf{f}$, and adds an edit or synchronous move if the current events e_i and f_j share the same id (so f_j stems from a trace event e_i), a model move if the id of the model event f_j does not occur in $\mathbf{e}$, and otherwise a log move. (For an event $e = (\iota, a, \alpha)$, we write $e.id$ to refer to ι.) In Line 6, the

Algorithm 4.1: Extracting an alignment from a state

Require: State $S = \langle E, C \rangle$, trace $\mathbf{e} = \langle e_0, \ldots, e_{n-1} \rangle$
Ensure: Alignment for $\mathbf{e}$

$model \leftarrow$ SMT model of formula $\bigwedge C$
$\langle f_0, \ldots, f_{m-1} \rangle \leftarrow$ sort events in E by assignment to ordering conditions in $model$
$moves \leftarrow []$, $i \leftarrow 0$, $j \leftarrow 0$
while $(i < n) \vee (j < m)$ **do**
  if $(i < n) \wedge (j < m) \wedge (e_i.id = f_j.id)$ **then**
    $moves.append(editOrSynchronousMove(e_i, f_j))$
    $i \leftarrow i + 1, j \leftarrow j + 1$
  else if $(j < m)$ and $f_j.id$ does not occur in $\mathbf{e}$ **then**
    $moves.append(modelMove(f_j))$
    $j \leftarrow j + 1$
  else
    $moves.append(logMove(e_i))$
    $i \leftarrow i + 1$
end while
return $moves$

number of mismatching assignments in e_i and f_j determines whether the move is an edit or synchronous move. Note that the alignment extracted from a state is in general not unique as there can be multiple SMT models. For instance, by applying Alg. 4.1 to state S_2 resp. state S_6 in Figure 4.2, one obtains the alignments γ_1 resp. γ_2 shown in Example 4.2.

Our correctness result below shows that the alignment extracted from S is optimal with cost K (cf. the proof in Appendix B). Our running example illustrates this result: in Figure 4.2, the goal state with minimal cost is S_2, and indeed the optimal alignment γ_1 is extracted from it (cf. Example 4.2).

Theorem 4.1: Correctness

If S is a goal state with minimal cost K in a search space for $\mathcal{M}$ and $\mathbf{e}$ then γ returned by Alg. 4.1 on input S and $\mathbf{e}$ is an optimal alignment of $\mathbf{e}$ wrt. $\mathcal{M}$ with cost K.

4.4 Implementation

Our approach is implemented in the tool DADA, developed in Kotlin, which interfaces with the SMT solvers Z3 [120] and Yices [121] as backends. These SMT solvers are used to evaluate

the satisfiability of logical constraints that blend both control-flow and data perspectives. The input model format is backward compatible with the one proposed in [28]. However, significant improvements have been made to the syntax for data conditions. In particular, it now supports the full expressiveness of the SMT-LIB2 language [55], allowing users to write complex Boolean formulas directly within DECLARE constraints. This enhancement broadens the types of data-aware behavioral patterns that can be specified and reasoned about. Furthermore, the cost function used during alignment computation is customizable. Users can specify distinct costs for moves in the log, the model, and edit operations, providing fine-grained control over the alignment behavior. Upon execution, the tool generates an optimal alignment, which can be output in either a human-readable format, similar to that illustrated in Example 4.2, or a compact machine-readable format. Additionally, the search space discovered during the computation can be exported as a graph, akin to the one depicted in Figure 4.2.

The DECLARE constraints language supported by our approach is, in fact, more expressive than initially introduced in Section 4.2. Specifically, branching in DECLARE constraints is enabled, as described in Chapter 3, and all DECLARE templates listed in Table 3.2 have been implemented.

Example 4.5: Illustration of a complex data constraint

Consider a shipping scenario with a DECLARE `Response` constraint between events *Package Shipment* (A) and *Delivery Confirmation* (B): once a shipment occurs, confirmation must follow. The constraint incorporates a data condition: if the package weighs under 10.5 kg and is destined for a specified geographic region, confirmation must arrive within 3 days of shipment; otherwise, the deadline extends to 10 days. Our solution handles this in SMT-LIB2 [55] notation or simply as $(A.weight < 10.5 \text{ and } B.region = \text{“}Europe\text{”})? (B.time - A.time \leq 3d) : (B.time - A.time \leq 10d)$. Note that we treat timestamps as data attributes in the event log and enforce control-flow constraints directly over them, thereby naturally covering quantitative time requirements like those in this example.

Encoding. The SMT solver reasons on control flow and data dependencies in tandem, to identify violations and possible repairs for each constraint. We thus need to check satisfiability of formulas that mix ordering and data conditions. Data conditions as in Definition 4.1, but also much richer conditions, can be directly expressed in SMT-LIB2. Ordering conditions on a

set E are encoded as follows: for every event $e \in E$ we use the SMT variable τ_e of integer type that encodes the event's timestamp. Then an ordering constraint $e_1 < e_2$ is directly translated to $\tau_{e_1} < \tau_{e_2}$; $\mathit{first}(e)$ is translated to $\bigwedge_{e' \in E \setminus \{e\}} \tau_e < \tau_{e'}$ and similar for $\mathit{last}(e)$; and $e \ll e'$ is translated to $\tau_{e_1} < \tau_{e_2} \wedge \bigwedge_{e \in E \setminus \{e_1, e_2\}} (\tau_e > \tau_{e_2} \vee \tau_e < \tau_{e_1})$. A constraint $e_1 <^a_{[c]} e_2$ is translated to $\tau_{e_1} < \tau_{e_2} \wedge \bigwedge_{e \in E_a} (\neg [c](e) \vee \tau_e > \tau_{e_2} \vee \tau_e < \tau_{e_1})$, where E_a is the set of all events in E with activity a, with e_1 and e_2 excluded. Moreover, for efficiency,

Optimizations. The following are the most influential implemented optimizations.

SMT assumptions. We use the SMT solver's assumption mechanism to temporarily check conditions, such as those in Definitions 4.10 and 4.11. This approach allows the algorithm to assert temporary assumptions on top of a core set of formulas that are needed multiple times while evaluating each state, enabling the solver to reuse previously learned information from the core context. As a result, this optimization avoids re-solving the entire formula from scratch for each check, significantly reducing redundant computation and improving performance.

Selecting a violation to repair. After identifying violations in the current state, the next step is to determine which violation to address first. These can be processed independently as all possible repairs are suggested when generating actions. The choice of the violation to repair next can significantly affect performance. The current approach involves selecting the violation resulting in the fewest child states, to delay the state explosion by creating very few branches toward the start of the search.

Early detection of dead-ends. It is worth noting that all violated activations are precomputed in order to apply the previous optimization, even if only one is selected for immediate repair. This is leveraged by the this optimization as it allows for the early detection of dead-ends. Dead-ends occur when no violation of a state can be repaired despite the presence of other violated activations that can be repaired. In these cases, there is no viable way to repair one of the errors, so it does not make sense to keep trying to repair other activations since it will not be possible to repair all of them. In case no repair is applicable for some violation, the state is a dead end, and the whole branch that shares this ancestor can be pruned from the search space.

Example 4.6: Early detection of dead-ends

Consider a state where two activations are violated: one for event e_1, which violates a control-flow constraint requiring that event e_2 must not occur after e_1, and another for

event e_3, which violates the data constraint $e_3.x \geq 0$. Suppose the data violation can be repaired by updating the value of x, but the control-flow violation cannot be resolved because e_2 was added in this branch to repair a previous constraint violation and is now required to appear after e_1. Removing e_2 to fix the violation with e_1 would invalidate prior repairs, making the violation unrecoverable, so there are no possible ways to repair the violation on e_1. In this case, even though one activation is still repairable, the other is not, rendering the state a dead-end even if it is technically possible to keep applying repairs. This situation is detected early, allowing the entire branch rooted at this state to be pruned from the search space without exploring it further.

Pruning of unsatisfiable states. If a state $S = \langle E, C \rangle$ was generated where $\bigwedge C$ is unsatisfiable, conflicting conditions were added while generating the state. Therefore, the state can be dropped from the search space.

Example 4.7: Pruning of unsatisfiable states

Consider a state where repairing a violation requires enforcing a data constraint on event e_1, such that $e_1.x \geq 0$. However, due to a previous repair applied earlier in the same branch, the same attribute x was already constrained to $e_1.x < 0$. As a result, the state accumulates conflicting data constraints over $e_1.x$, making them mutually unsatisfiable. In this case, the state cannot represent a valid solution, and it is pruned from the search space to avoid exploring infeasible continuations.

4.5 Evaluation

All algorithms were evaluated under identical conditions, namely a Java Virtual Machine[2] on an Intel 5220R CPU with 8 GB of RAM. All algorithms were executed in single-threaded mode. The source code, executable, dataset and raw results are publicly available.[3]

Dataset. The evaluation utilizes a synthetic dataset originally introduced in [28], specifically designed to facilitate systematic and scalable evaluations of alignment algorithms under

[2] OpenJDK 64-Bit VM Temurin 21.0.6+7-LTS.

[3] `https://apps.citius.gal/dada` and `https://doi.org/10.5281/zenodo.15470077`

increasing levels of complexity. The complexity of the process models is influenced by the number of constraints (3, 5, 7, or 10) and constraint modifications (replacing 0, 1, 2, or 3 constraints). For each model, multiple event logs with varying trace lengths were generated (10, 15, 20, 25, or 30 events), resulting in 68,000 trace-model pairs.

Each constraint in the models includes data conditions on both activation and target events. These conditions are restricted to simple expressions involving variable-to-constant comparisons over two types of attributes: $categorical$ attributes with three possible values ($c1$, $c2$, $c3$), and $integer$ attributes ranging from 0 to 100. For example, constraints may require that an activation event has an $integer$ attribute greater than 10, or a $categorical$ attribute equal to $c1$.

Performance comparison. We compare DADA, using either the Z3 [120] SMT solver or the Yices [121] SMT solver, to Bergami2021 [28], using the original SymBA* [122] planner or the Fast Downward [123] planner. Our experiments measure the execution time for each pair of model and trace. To ensure a rigorous comparison, we verified that the alignment costs produced by DADA-Z3 and DADA-Yices match those generated by Bergami2021-BA and Bergami2021-FD.

As shown in Figure 4.3, Data-Aware DECLARE Aligner exhibits significantly better scalability under increasing model and trace complexity. The number of alignments completed within a given time limit is consistently higher across all time thresholds. Both DADA-Z3aand DADA-Yices achieve a robust performance profile, completing every alignment in under 5 seconds. On average, DADA-Z· is 2.9 times faster than the Bergami2021-FD and 5.9 times faster than Bergami2021-BA. These results indicate that the combination of symbolic reasoning over control-flow and data, together with the optimizations introduced in Section 4.4, leads to more efficient search and alignment of data-aware DECLARE models. It is worth emphasizing that this fair comparison with prior work only captures a subset of the capabilities of the proposed method.

Constraint flexibility. While the previous experiment was limited to the data conditions supported by [28], our approach can leverage the power of SMT solvers to define complex data dependencies. To demonstrate the expressive power of Data-Aware DECLARE Aligner beyond prior work, we introduce complex correlation constraints involving arithmetic, modulo, and conditional operations across event attributes —constructs not supported in previous tools. In

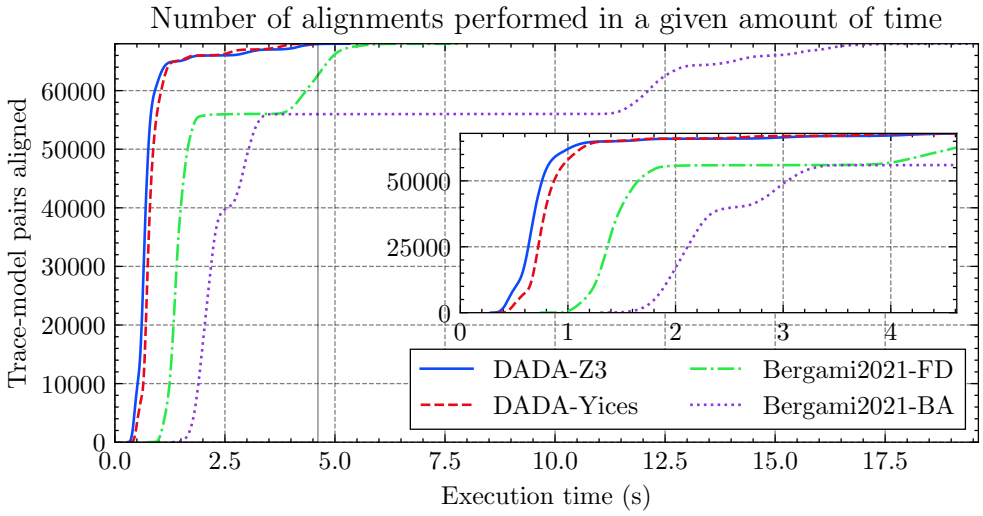


Figure 4.3: Number of trace-model pairs aligned within a given time frame by each algorithm. The enlargement on the right highlights the differences for shorter time intervals.

these conditions, A refers to the activation and T to the target; cat is an abbreviation for the *categorical* attribute, and $timestamp$ is the event's time.

(C1) $A.timestamp + A.integer * 1d > T.timestamp + T.integer * 1d$

(C2) $(A.cat - \text{“0”}) \,\%\, 10 < (T.cat - \text{“0”}) \,\%\, 10$

(C3) $(A.cat == T.cat)\ ?\ (T.cat \,\%\, 2 == 0) : (A.integer > T.integer)$

We generate a new dataset by augmenting models with combinations of the previous correlation constraints (negations, disjunctions, and conjunctions) while retaining the original activation and target conditions from the state of the art. These additions stress-test the solver by introducing cross-event arithmetic and logical dependencies, resulting in models like:

$$\texttt{Response}[activity\ 1,\ activity\ 2] \mid \ldots \mid \ldots \mid \neg(\neg(C1)\ or\ \neg(\neg(C3)\ and\ \neg(C2))) \mid$$

$$\texttt{Chain Response}[activity\ 3,\ activity\ 4] \mid \ldots \mid \ldots \mid \neg(C1)\ or\ \neg(C2)\ or\ (C3) \mid$$

The added correlation conditions make the alignment problem even harder by potentially increasing (i) the number of repairs required to reach the optimal alignment, (ii) the number

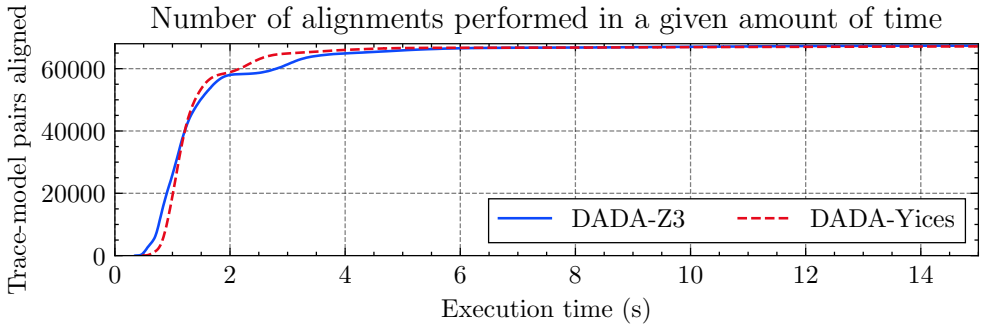


Figure 4.4: Performance evaluation incorporating correlation constraints.

of ways in which it is possible to repair them, and (iii) the work performed by the SMT solver within each state. To compensate, the models were simplified by only considering the ceiling of half of their constraints.

Despite increased complexity caused by the added correlation conditions, Figure 4.4 shows that our approach can handle them, with only a small percentage of alignments timing out after 5 minutes or running out of memory (0.06% for DADA-Yices and 0.91% for DADA-Z3).

4.6 Discussion

In this chapter, we have presented a novel approach to compute optimal alignments between event logs and data-aware declarative process models, leveraging the strengths of SMT solvers to handle complex data dependencies. Our approach combines A* search with SMT-based reasoning, enabling efficient exploration of the search space and identification of optimal alignments that satisfy both control flow and data conditions.

The key contributions of this work are threefold. Firstly, a new encoding scheme has been developed, which translates the control flow aspects of declarative process models into SMT formulas, facilitating efficient analysis. Secondly, the approach has been extended to handle multi-perspective process models, incorporating data conditions that can be expressed using standard SMT formulas and reasoning about control flow and data conditions at the seame time. Lastly, an A*-based search strategy has been proposed, which leverages the power of SMT solvers to identify and resolve constraint violations, ultimately finding the optimal alignment

between event logs and declarative process models. The correctness of the proposed algorithm is proven.

The evaluation of our approach has demonstrated its performance advantages over current state-of-the-art methods, achieving significantly faster execution times. In addition, our approach handles much more expressive data conditions that relate data in activation and target events, using arbitrary constraints supported by SMT solvers and hence handling more much expressive data conditions. Furthermore, our approach has shown improved scalability for large and complex process models, making it a viable solution for real-world applications.

Future research directions include optimizations such as advanced pruning strategies that leverage the SMT formula associated with each state, as well as strategic selection of which violation to address first for any given state, and the development of a heuristic function for the A* search.

Chapter 5

Conclusions

The primary objectives of this PhD thesis were to develop innovative algorithms for computing optimal alignments between event logs and procedural, declarative, and data-aware declarative process models. To achieve these objectives, this research has introduced a comprehensive framework for efficient alignment-based conformance checking, combining formal foundations with complete implementations and extensive evaluation on real and synthetic datasets. The results demonstrate clear improvements over the state of the art in both accuracy and performance. The following paragraphs detail the key contributions and empirical findings that substantiate these claims, highlighting the specific algorithmic innovations, optimization techniques, and comparative evaluations that collectively define the impact and novelty of this research.

Efficient optimal alignment algorithm for procedural process models. By applying a novel heuristic called Model Move Required (MMR) and leveraging A* search and optimization techniques, it was found that the performance of optimal alignment computation of Petri nets and event logs can be significantly improved, surpassing the performance standards set by the state-of-the-art methods in this highly-competitive domain. Notably, the MMR heuristic balances the time required to compute the heuristic with the accuracy of the estimates it provides, allowing for a more efficient exploration of the search space. This is achieved by finding a subset of the required transitions in the model and comparing their labels to the remaining activities in the trace, which enables the algorithm to focus on the most promising areas of the search space. Furthermore, the use of a partial reachability graph (PRG) was found to be highly effective in reducing the number of model operations required during the search, as

it allows the algorithm to quickly access enabled transitions and new markings without having to recompute them from scratch, reusing this information across the alignments of all traces of an event log. Additionally, state reduction-based optimizations, such as States Reduction forcing asynchronous Model moves (SRModel) and States Reduction forcing asynchronous Log moves (SRLog), were found to contribute to the performance of the algorithm by reducing the number of generated neighbors and mitigating the state explosion that occurs for complex datasets.

The optimizations proposed in this study were thoroughly evaluated through an ablation study, which examined their impact both individually and in combination with other optimizations. Notably, when the SRLog, SRModel, and PRG optimizations are enabled concurrently, the average execution time is reduced by 11.9% for simpler problems and a substantial 40.1% for more complex ones, thereby demonstrating the scalability of our approach. Furthermore, the number of states that need to be explored to reach an optimal solution undergoes a similar reduction, with decreases of 24.0% for simpler problems and 32.4% for more complex ones. In addition to the ablation study, another set of experiments was conducted to compare the performance of our algorithm, REACH, with the 10 most promising and recent tools from the state of the art. The results unequivocally show that REACH possesses a clear performance advantage, achieving an average rank of 3.242 while the next best algorithm from the state of the art ranks as 4.641. Moreover, REACH is capable of computing optimal alignments for 26% more log-model pairs than any other state-of-the-art tool in just 10 seconds. When the time limit is extended, REACH aligns 216 log-model pairs in a remarkably short timeframe of just 45 seconds, whereas the next best tool from the state of the art requires 5 minutes to align 2 fewer log-model pairs, underscoring the significant performance gap between REACH and existing state-of-the-art solutions.

Novel algorithm for optimal alignments in declarative process models. By reformulating the problem of optimal alignment of DECLARE process models and event logs to focus on constraint violation repair, it was found that a more efficient algorithm can be developed. Furthermore, the proposed DeclareAligner algorithm uses a heuristic function that leverages knowledge about required repair actions to provide an accurate estimate of the remaining cost to reach a goal state. This is achieved by identifying all violated activations in the current state and retrieving the set of proposed repair actions for each violation, which enables the algorithm to focus on the most promising areas of the search space. The algorithm also employs early

pruning techniques, such as detecting dead-ends and pruning branches that cannot lead to a the optimal state, to improve search efficiency. For example, if a violation has no applicable repair action, the algorithm can prune the current branch from the search space, as it is impossible to reach a goal state from that point. Furthermore, the use of constraint preprocessing techniques, such as merging nodes in the LTGraph of the initial state, was found to reduce problem complexity and improve efficiency by eliminating unnecessary states from the search space.

The reformulation of the optimal alignments problem is almost competitive with the current state of the art as shown by our evaluation section. However, the proposed reformulation opens up novel optimization techniques that were not previously possible and improve performance much further. This is investigated in the ablation study, which shows how enabling all the proposed optimizations results in a reduction of average execution time of an impressive 92.8%, accompanied by an average reduction on the number of explored states of 97.0%. When compared against the state of the art, DeclareAligner achieves an average execution time of 7.6 seconds while the next best algorithm requires 46.2 seconds on average. This results in an average rank of 1.24 for the proposed algorithm while the next best algorithm has a rank of 2.73, clearly identifying DeclareAligner as the top-performing algorithm. A notable achievement of DeclareAligner is its ability to align 90% of the tested trace-model pairs in 3.5 seconds or less per alignment, outperforming the next best state-of-the-art algorithm which requires up to 100 seconds per problem to reach the same number of solutions. Moreover, the proposed optimizations enable DeclareAligner to successfully compute alignments for 231 trace-model pairs that remain unsolved by all other algorithms within a 5-minute time limit.

Advanced conformance checking for data-aware declarative process models. By reimagining the DeclareAligner core algorithm to support rich data conditions while also integrating efficient SMT solvers into the core algorithm, it was found that an efficient and more expressive algorithm for computing optimal alignments between event logs and data-aware DECLARE can be developed. The Data-Aware DECLARE Aligner algorithm integrates A* search and SMT solving techniques to handle control-flow and data constraints simultaneously, which enables it to reason about complex data conditions such as deadlines and resource constraints. This is achieved by leveraging the strengths of both A* search and SMT solving, allowing the algorithm to efficiently explore the search space while ensuring that all data constraints are satisfied. The algorithm also employs innovative techniques, such as declaring a time attribute to enforce control-flow constraints directly over data, which allows for the specification of

complex time-based conditions. Additionally, the use of SMT-based dead-end detection was found to contribute to the performance of the algorithm by eliminating unnecessary states from the search space and reducing the number of applicable repairs that need to be considered. The algorithm also supports full SMT-LIB2 syntax, allowing for even more flexible constraints to be specified, such as constraints involving arithmetic operations or complex logical formulas.

The proposed approach has demonstrated significant performance advantages over current state-of-the-art methods. Notably, Data-Aware DECLARE Aligner achieves an average speedup of 2.9 times compared to Bergami2021-FD and 5.9 times compared to Bergami2021-BA. This substantial improvement in execution time is a direct result of our efficient SMT-based encoding scheme and A*-based search strategy, which enables the computation of optimal alignments that satisfy both control flow and data conditions. The evaluation results also highlight the unprecedented flexibility and expressiveness of our approach, as evidenced by its ability to handle complex data dependencies and correlation conditions that no other state-of-the-art algorithm is capable of handling.

Overall, the key findings of this thesis demonstrate the efficiency of the proposed algorithms in computing optimal alignments between event logs and procedural, declarative, and data-aware declarative process models. The results have significant implications for the field of Process Mining, enabling organizations to gain deeper insights into their processes, identify areas for improvement, and optimize their operations with greater confidence.

5.1 Limitations and future work

No research is ever entirely complete, and this work is no exception. While the contributions made throughout this PhD are significant, several limitations remain. These aspects are left as future work, offering exciting possibilities for continued exploration.

One key limitation lies in the scalability of the proposed approaches when applied to exceptionally large and complex models and event logs. Specifically, models with hundreds or thousands of Petri net transitions or DECLARE constraints, combined with long traces comprised of hundreds of events, can lead to an explosion of intermediate states during the search due to the vast number of potential alignment paths. Although the proposed algorithms are efficient, handling such complexity remains a challenge. Future research should investigate how these methods perform on large-scale benchmarks and explore advanced strategies, such as improved pruning techniques or parallelization, to further enhance efficiency.

Another area for improvement that holds great promise is the development of a data-aware solution for procedural process models. This PhD prioritized data-aware declarative process models. However, there is potential to build on the proposed REACH algorithm and adapt some of the optimizations from Data-Aware DECLARE Aligner to suit procedural models as well. In particular, integrating the SMT-based approach used in Data-Aware DECLARE Aligner to accommodate the specific characteristics of procedural models could result in more performant data-aware alignment techniques for procedural models.

Similarly, there is potential to enrich the Data-Aware DECLARE Aligner algorithm by incorporating ideas from the DECLAREALIGNER algorithm. In particular, leveraging SMT solvers within DECLARE process models could lead to more powerful intermediate state analysis and more effective state-space reduction. Furthermore, since DECLAREALIGNER does not consider data, it may be feasible to apply more aggressive optimizations that make assumptions about the input models, leading to further performance gains.

While Data-Aware DECLARE Aligner represents a strong step forward, it also presents a few limitations worth exploring. Firstly, the current algorithm for the selection of which violation to repair first for a given state is relatively naive: it simply selects the violation with the fewest child states. Although this is effective, exploring more sophisticated selection strategies could yield significant performance gains. Another limitation of Data-Aware DECLARE Aligner is the lack of a heuristic function to guide the search process. This stands in contrast to the other proposed algorithms, where the use of heuristics significantly accelerated performance. Developing a suitable heuristic, possibly by running tests against the underlying SMT formulas to estimate the remaining cost, could help focus the search on more relevant paths, reduce computational effort, and improve performance without compromising optimality.

Building on the foundational framework in efficient optimal alignments, it is also possible to expand into related areas that rely heavily on robust conformance checking capabilities, such as predictive monitoring, concept drift detection, and process repair. Predictive monitoring models benefit from training on traces that closely follow the process model; optimal alignments help map observed behavior that violates the process model into valid model paths, improving training quality. In concept drift detection, alignments provide a fine-grained view of how recent behavior deviates from past norms, allowing for more precise identification of changes. Similarly, process repair techniques depend on accurate alignment information to pinpoint deviations and suggest meaningful improvements. The high-performance alignment techniques developed during this PhD could play an important role in enabling further exploration of these

areas, as they provide a strong foundation for accurate and efficient analysis of event logs and process models. By achieving significant improvements in performance, it becomes possible to unlock new opportunities for process mining, ultimately enabling organizations to gain deeper insights into their processes, identify areas for improvement, and optimize their operations with greater confidence.

Bibliography

[1] Josep Carmona, Boudewijn van Dongen, Andreas Solti, and Matthias Weidlich. *Conformance checking: relating processes and models*. Springer, November 2018.

[2] Borja Vázquez-Barreiros, Manuel Mucientes, and Manuel Lama. Prodigen: Mining complete, precise and minimal structure process models with a genetic algorithm. *Information Sciences*, 294:315–333, February 2015.

[3] Jayamini Ranaweera, Dan Weaving, Marco Zanin, Matthew Pickard, and Gregory Roe. Digitally optimizing the information flows necessary to manage professional athletes: A case study in rugby union. *Frontiers in Sports and Active Living*, 4, June 2022.

[4] Jayamini Ranaweera, Dan Weaving, Marco Zanin, and Gregory Roe. Identifying the current state and improvement opportunities in the information flows necessary to manage professional athletes: A case study in rugby union. *Frontiers in Sports and Active Living*, 4, 6 2022.

[5] Noah Dubois. Advancing intelligent systems through artificial intelligence in cloud infrastructure, process mining, and predictive healthcare analytics. *QIT Press-International Journal of Artificial Intelligence Research and Development (QITP-IJAIRD)*, 6(1):15–22, 2025.

[6] Jorge Munoz-Gama, Niels Martin, Carlos Fernandez-Llatas, Owen A Johnson, Marcos Sepúlveda, Emmanuel Helm, Victor Galvez-Yanjari, Eric Rojas, Antonio Martinez-Millana, Davide Aloini, et al. Process mining for healthcare: Characteristics and challenges. *Journal of Biomedical Informatics*, 127:103994, 2022.

[7] R Mans, M Schonenberg, Minseok Song, W Van Der Aalst, and P Bakker. *Application of Process Mining in Healthcare – A Case Study in a Dutch Hospital*, pages 425–438. Springer Berlin Heidelberg, 2008.

[8] Álvaro Rebuge and Diogo Ferreira. Business process analysis in healthcare environments: A methodology based on process mining. *Information Systems*, 37(2):99–116, 4 2012.

[9] Alexander Wouter Poncin, Mark Serebrenik, Van Den, and Brand. Process mining software repositories. In *Proceedings of the European Conference on Software Maintenance and Reengineering, (CSMR 2011)*, pages 5–14. IEEE Computer Society, 2011.

[10] M Artini, Lemos, C Caio, Ricardo Massa Ferreira Sabino, César Lima, and De Oliveira. Using process mining in software development process management: A case study. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, (SMC 2011)*, pages 1181–1186. IEEE Computer Society, 2011.

[11] Vladimir Rubin, Alexey Mitsyuk, Irina Lomazova, and Wil Van Der Aalst. Process mining can be applied to software too! In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, volume 57, pages 1–8. ACM, 9 2014.

[12] Renata Gabryelczyk, Janice C Sipior, and Aneta Biernikowicz. Motivations to adopt bpm in view of digital transformation. *Information Systems Management*, 41(4):340–356, 2024.

[13] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo Reijers. *Fundamentals of Business Process Management*. Springer Berlin Heidelberg, 2018.

[14] Wil M. P. van der Aalst, Arya Adriansyah, Ana Karla Alves de Medeiros, Franco Arcieri, Thomas Baier, Tobias Blickle, and Bose, J.C. et al. Process Mining Manifesto. In *Business Process Management 2011 International Workshops*, pages 169–194, 2012.

[15] Wil van der Aalst. In *Process Mining - Data Science in Action*. Springer, 2016.

[16] Alifah Syamsiyah, Boudewijn F van Dongen, and Wil MP van der Aalst. Recurrent process mining on procedural and declarative approaches. *BPM Center Report BPM-17-03*, 2017.

[17] Luise Pufahl, Francesca Zerbato, Barbara Weber, and Ingo Weber. Bpmn in healthcare: Challenges and best practices. *Information Systems*, 107:102013, 2022.

[18] Guanjun Liu. *Petri Nets: Theoretical Models and Analysis Methods for Concurrent Systems*. Springer Nature, 2022.

[19] Cornelia Haisjackl, Irene Barba, Stefan Zugal, Pnina Soffer, Irit Hadar, Manfred Reichert, Jakob Pinggera, and Barbara Weber. Understanding declare models: strategies, pitfalls, empirical results. *Software & Systems Modeling*, 15(2):325–352, 2016.

[20] David Chapela-Campa, Marlon Dumas, Manuel Mucientes, and Manuel Lama. Efficient edge filtering of directly-follows graphs for process mining. *Information Sciences*, 610:830–846, 2022.

[21] T Jouck, B Depaire, and Ptandloggenerator. In *A generator for artificial event data Proceedings of the BPM Demo Track 2016 Co-Located with the 14th International Conference on Business Process Management (BPM 2016)*, volume 1789, pages 23–27, Rio de Janeiro, Brazil, 9 2016.

[22] Alexander Gebharter. Causal nets, interventionism, and mechanisms. *Cham: Springer*, 2017.

[23] Yibin Xu, Tijs Slaats, Boris Düdder, Thomas Troels Hildebrandt, and Tom Van Cutsem. Safe design and evolution of smart contracts using dynamic condition response graphs to model generic role-based behaviors. *Journal of Software: Evolution and Process*, 37(1):e2730, 2025.

[24] Michael Blondin, Filip Mazowiecki, and Philip Offtermatt. Verifying generalised and structural soundness of workflow nets via relaxations. In *International Conference on Computer Aided Verification*, pages 468–489. Springer, 2022.

[25] Claudio Di Ciccio and Marco Montali. Declarative process specifications: reasoning, discovery, monitoring. In *Process mining handbook*, pages 108–152. Springer International Publishing Cham, 2022.

[26] Stijn Goedertier, Jan Vanthienen, and Filip Caron. Declarative business process modelling: principles and modelling languages. *Enterprise Information Systems*, 9(2):161–185, 2015.

[27] Felix Mannhardt. Multi-perspective process mining. 2018.

[28] Giacomo Bergami, Fabrizio Maria Maggi, Andrea Marrella, and Marco Montali. *Aligning Data-Aware Declarative Process Models and Event Logs*, pages 235–251. Springer International Publishing, 2021.

[29] Sebastian Dunzer, Matthias Stierle, Martin Matzner, and Stephan Baier. Conformance checking: a state-of-the-art literature review. In *Proceedings of the 11th international conference on subject-oriented business process management*, pages 1–10, 2019.

[30] Alessandro Berti and Wil M. P. van der Aalst. A novel token-based replay technique to speed up conformance checking and process enhancement. *Transactions on Petri Nets and Other Models of Concurrency*, 15:1–26, 2021.

[31] Anne Rozinat and Wil MP Van der Aalst. Conformance checking of processes based on monitoring real behavior. *Information Systems*, 33(1):64–95, 2008.

[32] A. Adriansyah. *Aligning observed and modeled behavior*. PhD thesis, Technische Universiteit Eindhoven, 2014.

[33] Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[34] Jacobo Casas-Ramos, Manuel Mucientes, and Manuel Lama. REACH: Researching Efficient Alignment-based Conformance Checking. *Expert Systems with Applications*, 241:122467, 2024.

[35] Jacobo Casas-Ramos, Manuel Lama, and Manuel Mucientes. DeclareAligner: A Leap Towards Efficient Optimal Alignments for Declarative Process Model Conformance Checking. *Engineering Applications of Artificial Intelligence*, 2025.

[36] Jacobo Casas-Ramos, Sarah Winkler, Alessandro Gianola, Marco Montali, Manuel Mucientes, and Manuel Lama. Efficient conformance checking of rich data-aware declare specifications. In *Business Process Management*, pages 88–105, Cham, 2026. Springer Nature Switzerland.

[37] Massimiliano de Leoni, Giacomo Lanciano, and Andrea Marrella. Aligning partially-ordered process-execution traces and models using automated planning. In *28th*

International Conference on Automated Planning and Scheduling (ICAPS), pages 321–329, 2018.

[38] Xixi Lu, Dirk Fahland, and Wil M. P. van der Aalst. Conformance checking based on partially ordered event data. In *Business Process Management 2014 Workshops*, pages 75–88. Lecture Notes in Business Information Processing, 2015.

[39] Massimiliano de Leoni and Wil M. P. van der Aalst. Aligning event logs and process models for multi-perspective conformance checking: An approach based on integer linear programming. In *11th International Conference on Business Process Management (BPM)*, pages 113–129. Springer, 2013.

[40] Farbod Taymouri and Josep Carmona. A recursive paradigm for aligning observed behavior of large structured process models. In *14th International Conference Business Process Management (BPM)*, pages 197–214. 2016.

[41] Boudewijn F. van Dongen. Efficiently computing alignments. In *Business Process Management*, pages 197–214, Cham, 2018. Springer International Publishing.

[42] Massimiliano de Leoni and Andrea Marrella. Aligning real process executions and prescriptive process models through automated planning. *Expert Systems with Applications*, 82:162–183, 2017.

[43] Maja Pesic, Helen Schonenberg, and Wil M. P. van der Aalst. Declare: Full support for loosely-structured processes, 2007.

[44] Massimiliano de Leoni, Fabrizio Maria Maggi, and Wil M. P. van der Aalst. Aligning event logs and declarative process models for conformance checking. In *Lecture Notes in Computer Science*, pages 82–97. Springer Berlin Heidelberg, 2012.

[45] Boudewijn F. van Dongen, Johannes De Smedt, Claudio Di Ciccio, and Jan Mendling. Conformance checking of mixed-paradigm process models. *Information Systems*, 102:101685, December 2021.

[46] Giuseppe De Giacomo, Francesco Fuggitti, Fabrizio Maria Maggi, Andrea Marrella, and Fabio Patrizi. A tool for declarative trace alignment via automated planning. *Softw. Impacts*, 16:100505, 2023.

[47] Paul Gastin and Denis Oddoux. Fast ltl to büchi automata translation. In *Computer Aided Verification: 13th International Conference, CAV 2001 Paris, France, July 18–22, 2001 Proceedings 13*, pages 53–65. Springer, 2001.

[48] Andrea Burattin, Fabrizio M Maggi, and Alessandro Sperduti. Conformance checking based on multi-perspective declarative process models. *Expert systems with applications*, 65:194–211, dec 2016.

[49] Felix Mannhardt, Massimiliano de Leoni, Hajo Reijers, and Wil van der Aalst. Balanced multi-perspective checking of process conformance. *Computing*, 98(4):407–437, 2016.

[50] Paolo Felli, Alessandro Gianola, Marco Montali, Andrey Rivkin, and Sarah Winkler. Cocomot: Conformance checking of multi-perspective processes via SMT. In *Proc. of BPM 2021*, volume 12875 of *LNCS*, pages 217–234. Springer, 2021.

[51] Fabrizio Maria Maggi, Andrea Marrella, Fabio Patrizi, and Vasyl Skydanienko. Data-aware declarative process mining with sat. *ACM Transactions on Intelligent Systems and Technology*, 14(4):1–26, August 2023.

[52] Axel Kjeld Fjelrad Christfort and Tijs Slaats. Efficient optimal alignment between dynamic condition response graphs and traces. In *Business Process Management*, page 3–19, Berlin, Heidelberg, 2023. Springer-Verlag.

[53] Massimiliano de Leoni, Fabrizio M. Maggi, and Wil M.P. van der Aalst. An alignment-based framework to check the conformance of declarative process models and to preprocess event-log data. *Information Systems*, 47:258–277, January 2015.

[54] Giuseppe De Giacomo, Fabrizio Maria Maggi, Andrea Marrella, and Fabio Patrizi. On the disruptive effectiveness of automated planning for ltlf-based trace alignment. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1), feb 2017.

[55] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB standard: Version 2.7. Technical report, 2025.

[56] Wil M. P. van der Aalst and Josep Carmona, editors. *Process Mining Handbook*, volume 448 of *LNBIP*. Springer, 2022.

[57] Marlon Dumas, Fabiana Fournier, Lior Limonad, Andrea Marrella, Marco Montali, Jana-Rebecca Rehse, Rafael Accorsi, Diego Calvanese, Giuseppe De Giacomo, Dirk

Fahland, Avigdor Gal, Marcello La Rosa, Hagen Völzer, and Ingo Weber. Augmented business process management systems: A research manifesto, 01 2022.

[58] Ederson Carvalhar Fernandes, Barry Fitzgerald, Liam Brown, and Milton Borsato. Machine learning and process mining applied to process optimization: Bibliometric and systemic analysis. *Procedia Manufacturing*, 38:84–91, 2019.

[59] Alessandro Berti, Sebastiaan van Zelst, and Daniel Schuster. Pm4py: A process mining library for python. *Software Impacts*, 17:100556, 2023.

[60] Henrik Kaemmerling, Eduardo Goulart Rocha, and Wil MP van der Aalst. Prom4py-a python wrapper for the prom framework. 2024.

[61] Yutika Amelia Effendi and Riyanarto Sarno. Parallel process discovery using a new time-based alpha++ miner. *IIUM Engineering Journal*, 21(1):126–141, 2020.

[62] Adriano Augusto, Raffaele Conforti, Marlon Dumas, Marcello La Rosa, and Artem Polyvyanyy. Split miner: automated discovery of accurate and simple business process models from event logs. *Knowledge and Information Systems*, 59:251–284, 2019.

[63] Ali Norouzifar, Marcus Dees, and Wil van der Aalst. Imposing rules in process discovery: An inductive mining approach. In *International Conference on Research Challenges in Information Science*, pages 220–236. Springer, 2024.

[64] Angelina Prima Kurniati, Guntur Kusuma, and Gede Wisudiawan. Implementing heuristic miner for different types of event logs. *International Journal of Applied Engineering Research*, 11(8):5523–5529, 2016.

[65] Indra Waspada, Riyanarto Sarno, Endang Siti Astuti, Hanung Nindito Prasetyo, and Raden Budiraharjo. Graph-based token replay for online conformance checking. *IEEE Access*, 10:102737–102752, 2022.

[66] Daniel Reißner, Abel Armas-Cervantes, Raffaele Conforti, Marlon Dumas, Dirk Fahland, and Marcello La Rosa. Scalable alignment of process models and event logs: An approach based on automata and S-components. *Information Systems*, page 101561, 2020.

[67] Wai Lam Jonathan Lee, H.M.W. Verbeek, Jorge Munoz-Gama, Wil M.P. van der Aalst, and Marcos Sepúlveda. Recomposing conformance: Closing the circle on decomposed

alignment-based conformance checking in process mining. *Information Sciences*, 466:55–91, oct 2018.

[68] D. Reißner, R. Conforti, M. Dumas, M. La Rosa, and A. Armas-Cervantes. Scalable conformance checking of business processes. In *On the Move to Meaningful Internet Systems: OTM 2017 Conferences*, pages 607–627, October 2017.

[69] Almir Djedovic, Emir Zunic, and Almir Karabegovic. *Model business process improvement by statistical analysis of the users' conduct in the process*, pages 1–6. IEEE, SpliTech, 7 2016.

[70] Dirk Fahland and Wil MP Van Der Aalst. Model repair—aligning process models to reality. *Information Systems*, 47:220–243, 2015.

[71] Armas Cervantes and A. *Process Model Repair*, pages 1316–1321. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics. Springer International Publishing, 2017.

[72] XES Working Group. IEEE standard for extensible event stream (XES) for achieving interoperability in event logs and event streams. *IEEE Std 1849-2023*, 2023.

[73] Heidy M Marin-Castro and Edgar Tello-Leal. Event log preprocessing for process mining: a review. *Applied Sciences*, 11(22):10556, 2021.

[74] Wil M. P. van der Aalst. Decomposing Petri nets for process mining: A generic approach. *Distributed and Parallel Databases*, 31(4):471–507, jul 2013.

[75] J.C.A.M. Buijs. Environmental permit application process ('wabo'), coselog project, 2014.

[76] Wil van der Aalst. Process mining: discovering and improving spaghetti and lasagna processes. In *2011 IEEE symposium on computational intelligence and data mining (CIDM)*, pages 1–7. IEEE, IEEE, 2011.

[77] Ioannis Mavroudopoulos, Christos Balaktsis, Konstantinos Varvoutas, Georgia Kougka, Anastasios Gounaris, and Marco Comuzzi. Declarative process mining in big data scenarios using an application-agnostic framework. *Process Science*, 2(1):6, 2025.

[78] Alessandro Berti and Wil MP van der Aalst. Reviving token-based replay: Increasing speed while improving diagnostics. *ATAED@ Petri Nets/ACSD*, 2371:87–103, 2019.

[79] Sebastiaan J van Zelst, Joos CAM Buijs, Borja Vázquez-Barreiros, Manuel Lama, and Manuel Mucientes. Repairing alignments of process models. *Business & Information Systems Engineering*, 62:289–304, 2020.

[80] Michael Grohs, Peter Pfeiffer, and Jana-Rebecca Rehse. Business process deviation prediction: Predicting non-conforming process behavior. In *2023 5th International Conference on Process Mining (ICPM)*, pages 113–120, Oct 2023.

[81] Jiaojiao Wang, Dingguo Yu, Xiaoyu Ma, Chang Liu, Victor Chang, and Xuewen Shen. Online predicting conformance of business process with recurrent neural networks. In Gary Wills, Peter Kacsuk, and Victor Chang, editors, *IoTBDS 2020 - Proceedings of the 5th International Conference on Internet of Things, Big Data and Security*, IoTBDS 2020 - Proceedings of the 5th International Conference on Internet of Things, Big Data and Security, pages 88–100. SciTePress, May 2020.

[82] Jorge Munoz-Gama, Josep Carmona, and Wil M. P. van der Aalst. Single-entry single-exit decomposed conformance checking. *Information Systems*, 46:102–122, dec 2014.

[83] Mohammadreza Fani Sani, Sebastiaan J. van Zelst, and Wil M. P. van der Aalst. Conformance checking approximation using subset selection and edit distance. In *32nd International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 234–251. Springer, December 2020.

[84] Farbod Taymouri and Josep Carmona. An evolutionary technique to approximate multiple optimal alignments. In *16th International Conference Business Process Management (BPM)*, pages 215–232. 2018.

[85] Efrén Rama-Maneiro, Juan C Vidal, and Manuel Lama. Embedding graph convolutional networks in recurrent neural networks for predictive monitoring. *IEEE Transactions on Knowledge and Data Engineering*, 36(1):137–151, 2023.

[86] Víctor Gallego-Fontenla, Pedro Gamallo-Fernandez, Juan C Vidal, and Manuel Lama. Gradual drift detection in process models using conformance metrics. *ACM Transactions on Knowledge Discovery from Data*, 19(3):1–40, 2025.

[87] Josep Carmona, Boudewijn van Dongen, and Matthias Weidlich. *Conformance Checking: Foundations, Milestones and Challenges*, volume 448 of *LNBIP*, pages 155–190. Springer International Publishing, 2022.

[88] Seppe KLM van den Broucke, Jorge Munoz-Gama, Josep Carmona, Bart Baesens, and Jan Vanthienen. Event-based real-time decomposed conformance analysis. In *On the Move to Meaningful Internet Systems: OTM 2014 Conferences*, pages 345–363, 2014.

[89] Sander J.J. Leemans, Wil M. P. van der Aalst, Tobias Brockhoff, and Artem Polyvyanyy. Stochastic process mining: Earth movers' stochastic conformance. *Information Systems*, 102:101724, dec 2021.

[90] Luciano Garcia-Banuelos, Nick R. T. P. van Beest, Marlon Dumas, Marcello La Rosa, and Willem Mertens. Complete and interpretable conformance checking of business processes. *IEEE Transactions on Software Engineering*, 44(3):262–290, mar 2018.

[91] Farbod Taymouri and Josep Carmona. Computing alignments of well-formed process models using local search. *ACM Transactions on Software Engineering and Methodology*, 29(3):1–41, 2020.

[92] Boudewijn van Dongen, Josep Carmona, Thomas Chatain, and Farbod Taymouri. Aligning modeled and observed behavior: A compromise between computation complexity and quality. In *29th International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 94–109. Springer, 2017.

[93] Jorge Munoz-Gama, Josep Carmona, and Wil M. P. van der Aalst. Hierarchical conformance checking of process models based on event logs. In *34th International Conference Application and Theory of Petri Nets and Concurrency (PETRI NETS)*, pages 291–310. 2013.

[94] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Scalable process discovery and conformance checking. *Software & Systems Modeling*, 17(2):599–631, 2018.

[95] Daniel Reißner, Abel Armas-Cervantes, and Marcello La Rosa. Efficient conformance checking using alignment computation with tandem repeats. *arXiv preprint arXiv:2004.01781*, 2020.

[96] Wil M P van der Aalst. Structural characterizations of sound workflow nets. *Computing science reports*, 96(23):18–22, 1996.

[97] Reggie Davidrajuh. Extended reachability graph of Petri net for cost estimation. In *2013 8th EUROSIM Congress on Modelling and Simulation*, pages 378–383, sep 2013.

[98] Felix Mannhardt. Sepsis Cases - Event Log, 12 2016.

[99] L. Maruster, A. Weijters, W.M.P. van der Aalst, and A. van den Bosch. A rule-based approach for process discovery: Dealing with noise and imbalance in process logs. *Data Mining and Knowledge Discovery*, 13(1):67–87, 2006. Pagination: 21.

[100] Jorge Munoz-Gama, Josep Carmona, and Wil MP Van Der Aalst. Conformance checking in the large: Partitioning and topology. In *Business Process Management: 11th International Conference, BPM 2013, Beijing, China, August 26-30, 2013. Proceedings*, pages 130–145. Springer, 2013.

[101] Boudewijn van Dongen. BPI Challenge 2012, 4 2012.

[102] M. (Massimiliano) de Leoni and Felix Mannhardt. Road Traffic Fine Management Process, 2 2015.

[103] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Discovering block-structured process models from event logs containing infrequent behaviour. In *Business Process Management 2013 International Workshops*, pages 66–78. 2014.

[104] Arya Adriansyah, Jorge Munoz-Gama, Josep Carmona, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. Alignment based precision checking. In *Business Process Management 2012 International Workshops*, pages 137–149. 2013.

[105] Francesco Chiariello, Fabrizio Maria Maggi, and Fabio Patrizi. ASP-based declarative process mining. In *Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI 2022)*, pages 5539–5547. AAAI Press, 2022.

[106] Ivan Donadello, Francesco Riva, Fabrizio Maria Maggi, and Aladdin Shikhizada. Declare4Py: A Python library for Declarative Process Mining. In *Proceedings of the Best Dissertation Award, Doctoral Consortium, and Demonstration & Resources Track BPM 2022*, volume 3216 of *CEUR Workshop Proceedings*, pages 117–121. CEUR-WS.org, 2022.

[107] Fabrizio Maria Maggi, Marco Montali, and Ubaier Bhat. Compliance monitoring of multi-perspective declarative process models. In Remco Dijkman and Samira Si-Said, editors, *Proceedings of the 23rd IEEE International Enterprise Distributed Object Computing Conference (EDOC 2019)*, pages 151–160. IEEE Computer Society, oct 2019.

[108] Fabrizio Maria Maggi, Marco Montali, Michael Westergaard, and Wil M. P. van der Aalst. Monitoring business constraints with linear temporal logic: An approach based on colored automata. In *International Conference on Business Process Management*, pages 132–147. Springer Berlin Heidelberg, 2011.

[109] Marco Montali, Fabrizio Maggi, Federico Chesani, Paola Mello, and Wil Aalst. Monitoring business constraints with the event calculus. *ACM Transactions on Intelligent Systems and Technology*, 5(1):1–30, 12 2013.

[110] Francesco Riva, Dario Benvenuti, Fabrizio Maria Maggi, Andrea Marrella, and Marco Montali. An sql-based declarative process mining framework for analyzing process data stored in relational databases. In *International Conference on Business Process Management*, pages 214–231. Springer, 2023.

[111] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE Comput. Soc, 2001.

[112] Michael Westergaard. Better algorithms for analyzing and enacting declarative workflow languages using ltl. In *Business Process Management: 9th International Conference, BPM 2011, Clermont-Ferrand, France, August 30-September 2, 2011. Proceedings 9*, pages 83–98. Springer, 2011.

[113] Johannes De Smedt, Seppe K.L.M. Vanden Broucke, Jochen De Weerdt, and Jan Vanthienen. A full r/i-net construct lexicon for declare constraints. *SSRN Electronic Journal*, 2015.

[114] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[115] Claudio Di Ciccio, Mario Luca Bernardi, Marta Cimitile, and Fabrizio Maria Maggi. *Generating Event Logs Through the Simulation of Declare Models*, pages 20–36. Springer International Publishing, Cham, 2015.

[116] Paolo Felli, Marco Montali, Fabio Patrizi, and Sarah Winkler. Monitoring arithmetic temporal properties on finite traces. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence*, pages 6346–6354. AAAI Press, 2023.

[117] Zsuzsanna Nagy and Agnes Werner-Stark. An alignment-based multi-perspective online conformance checking technique. *Acta Polytechnica Hungarica*, 19(4):105–127, 2022.

[118] Stefan Schönig, Andreas Rogge-Solti, Cristina Cabanillas, Stefan Jablonski, and Jan Mendling. Efficient and customisable declarative process mining with SQL. In *Proceedings of the 28th International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 290–305, 2016.

[119] Diana Borrego and Irene Barba. Conformance checking and diagnosis for declarative business process models in data-aware scenarios. *Expert systems with applications*, 41(11):5340–5352, 2014.

[120] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science (LNCS)*, pages 337–340, 2008.

[121] Bruno Dutertre. Yices 2.2. In *Proceedings of the 14th Computer Aided Verification (CAV)*, pages 737–744, 2014.

[122] Alvaro Torralba, Vidal Alcázar, Daniel Borrajo, Peter Kissmann, and Stefan Edelkamp. SymBA*: A symbolic bidirectional A* planner. In *Planning Competition*, pages 105–108, 2014.

[123] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[124] Liz Allen, Alison O'Connell, and Veronique Kiermer. How can we ensure visibility and diversity in research contributions? how the contributor role taxonomy (credit) is helping the shift from authorship to contributorship. *Learned Publishing*, 32(1):71–74, January 2019.

APPENDIX A

SEMANTICS OF DATA-AWARE DECLARE

The following definition clarifies when a trace satisfies DECLARE constraints with data. For an assignment α with domain V, we write α^a for some $a \in \mathcal{A}$ for the same assignment on labeled variables, i.e., the assignment with domain $\{v^a \mid v \in V\}$ that sets $\alpha^a(v^a) = \alpha(v)$. Moreover, we write $\alpha \models c$ to express that α satisfies a condition c. For the union of two assignments α, β with disjoint domain we write $\alpha \cup \beta$.

Definition A.1: Data-Aware DECLARE Constraints

A constraint $\psi = \langle \varphi, c_{act}, c_{tgt}, c_{cor} \rangle$ is satisfied by a trace $\mathbf{e} = \langle e_0, \ldots, e_{m-1} \rangle$ if

- $\varphi = \texttt{Existence}(n, a)$, there are n distinct events $e_{i_1}, \ldots, e_{i_n}$ in $\mathbf{e}$ such that for all $1 \leq j \leq n$ and if e_{i_j} has the form $e_{i_j} = \langle \iota_j, a, \alpha_j \rangle$ it holds that $\alpha^a_j \models c_{tgt}$;
- $\varphi = \texttt{Absence}(n, a)$, and $\mathbf{e}$ does not satisfy $\langle \texttt{Existence}(n, a), c_{act}, c_{tgt}, c_{cor} \rangle$;
- $\varphi = \texttt{Init}(a)$, $e_0 = \langle \iota, a, \alpha \rangle$, and $\alpha^a \models c_{tgt}$;
- $\varphi = \texttt{End}(a)$, $e_{m-1} = \langle \iota, a, \alpha \rangle$, and $\alpha^a \models c_{tgt}$;
- $\varphi = \texttt{Choice}(a, b)$, and there is some $e_i = \langle \iota, d, \alpha \rangle$, $1 \leq i < m$, such that $d = a$ and $\alpha^a \models c_{tgt}$, or $d = b$and $\alpha^b \models c_{tgt}$;

- $\varphi = \texttt{RespondedExistence}(a,b)$, and either there is no $e_i = \langle \iota, a, \alpha \rangle$, $0 \leq i < m$, such that $\alpha^a \models c_{act}$, or there is some $e_j = \langle \iota', b, \beta \rangle$, with $0 \leq j < m$ and $i \neq j$, such that $\alpha^a \cup \beta^b \models c_{tgt} \wedge c_{cor}$;
- $\varphi = \texttt{Response}(a,b)$, and either there is no $e_i = \langle \iota, a, \alpha \rangle$, $0 \leq i < m$, such that $\alpha^a \models c_{act}$, or there is some $e_j = \langle \iota', b, \beta \rangle$, with $i < j < m$, such that $\alpha^a \cup \beta^b \models c_{tgt} \wedge c_{cor}$;
- $\varphi = \texttt{AlternateResponse}(a,b)$, and either there is no $e_i = \langle \iota, a, \alpha \rangle$, $0 \leq i < m$, such that $\alpha^a \models c_{act}$, or there is some $e_j = \langle \iota', b, \beta \rangle$, with $i < j < m$, such that $\alpha^a \cup \beta^b \models c_{tgt} \wedge c_{cor}$, and for all e_k with $i < k < j$ of the form $e_k = \langle \iota', d, \alpha_k \rangle$ either $d \neq a$ or $\alpha^a \not\models c_{act}$;
- $\varphi = \texttt{ChainResponse}(a,b)$, and either there is no $e_i = \langle \iota, a, \alpha \rangle$, $1 \leq i \leq n$, such that $\alpha^a \models c_{act}$, or $i < m-1$ and $e_{i+1} = \langle \iota', b, \beta \rangle$ and $\alpha^a \cup \beta^b \models c_{tgt} \wedge c_{cor}$;
- $\varphi = \texttt{Precedence}(a,b)$, and either there is no $e_i = \langle \iota, b, \alpha \rangle$, $0 \leq i < m$, such that $\alpha^b \models c_{act}$, or $i > 0$ and there is some $e_j = \langle \iota', a, \beta \rangle$, with $0 \leq j < i$, such that $\alpha^b \cup \beta^a \models c_{tgt} \wedge c_{cor}$;
- $\varphi = \texttt{AlternatePrecedence}(a,b)$, and either there is no $e_i = \langle \iota, b, \alpha \rangle$, $0 \leq i < m$, such that $\alpha^b \models c_{act}$, or $i > 0$ and there is some $e_j = \langle \iota', a, \beta \rangle$, with $0 \leq j < i$, such that $\alpha^b \cup \beta^a \models c_{tgt} \wedge c_{cor}$, and for all e_k with $j < k < i$ of the form $e_k = \langle \iota', d, \alpha_k \rangle$ either $d \neq b$ or $\alpha^b \not\models c_{act}$;
- $\varphi = \texttt{ChainPrecedence}(a,b)$, and either there is no $e_i = \langle \iota, b, \alpha \rangle$, $0 \leq i < m$, such that $\alpha^b \models c_{act}$, or $i > 0$ and $e_{i-1} = \langle \iota', a, \beta \rangle$ and $\alpha^b \cup \beta^a \models c_{tgt} \wedge c_{cor}$;
- $\varphi = \texttt{NotResponse}(a,b)$, and $\mathbf{e}$ does not satisfy $\langle \texttt{Response}(n,a), c_{act}, c_{tgt}, c_{cor} \rangle$;
- $\varphi = \texttt{NotRespondedExistence}(a,b)$, and $\mathbf{e}$ does not satisfy the constraint $\langle \texttt{RespondedExistence}(n,a), c_{act}, c_{tgt}, c_{cor} \rangle$; or
- $\varphi = \texttt{NotChainResponse}(a,b)$, and trace $\mathbf{e}$ does not satisfy the constraint $\langle \texttt{ChainResponse}(n,a), c_{act}, c_{tgt}, c_{cor} \rangle$.

APPENDIX B

PROOFS AND SUPPLEMENTARY MATERIAL OF DATA-AWARE DECLARE ALIGNER

Below, we denote by $\Gamma(S)$ the set of alignments that are possible results of Alg. 4.1 on input S. The following is straightforward to show by a case distinction on ψ.

Lemma B.1: Constraint violation

Let $\psi = (\varphi, c_{act}, c_{tgt}, c_{cor})$ be a constraint. A state S does not violate ψ if and only if for all $\gamma \in \Gamma(S)$, it holds that $\gamma|_M$ satisfies ψ.

We next show that if Alg. 4.1 returns γ on input S and $\mathbf{e}$ then γ is indeed an alignment for $\mathbf{e}$, though in general only wrt. a subset of $\mathcal{M}$, and the cost of γ coincides with $cost(S)$. Below, for a fixed trace $\mathbf{e}$, we denote by $\Gamma(S)$ the set of alignments that can be extracted from S, i.e., that are possible results of Alg. 4.1 on input S (recall that the alignment extracted from a state S is in general not unique).

Below, we make the following assumption (†) on the search space generated for $\mathcal{M}$ and $\mathbf{e}$: Repairs that remove an event e are only applied if e was never modified beforehand by a repair that changes the assignment.

Lemma B.2: Soundness

For each $\gamma \in \Gamma(S)$, it holds that $\gamma|_L = \mathbf{e}$ and $\kappa(\gamma) = cost(S)$.

Proof B.1: Lemma B.2

First of all, it is easy to see that if $S = \langle E, C \rangle$ and $\gamma \in \Gamma(S)$ then $\gamma|_L = \mathbf{e}$ because Alg. 4.1 adds for every e_i in $\mathbf{e}$ a log or synchronous/edit move, since i is incremented from 0 to $n-1$.

It remains to show the claim about the cost of γ, which we prove by induction on the depth n at which S occurs in the search tree. The statement holds for the initial state S_0, where an alignment $\gamma \in \Gamma(S)$ consists of only synchronous moves, so $\kappa(\gamma) = 0 = cost(S_0)$.

Let $S' = \langle E', C' \rangle$ be at depth $n+1$ in the search tree. The induction hypothesis is that the claim holds for all states at level n. We perform a case distinction on the repair applied at the parent $S = \langle E, C \rangle$ of S' to create S'.

- If an event e was added, then e has a fresh id that does not occur in the trace. Each alignment $\gamma' \in \Gamma(S')$ stems from some model μ for C'. By construction of the repair (the constraints remain the same), μ also satisfies C, giving rise to an alignment $\gamma \in \Gamma(S)$. By the induction hypothesis, γ is an alignment of $\mathbf{e}$ with cost $cost(S)$. Alignment γ' must be like γ except for an additional model move (as the added id is fresh, it cannot match an event in the trace), so that $\kappa(\gamma') = \kappa(\gamma) + 1$. Since we have $cost(S') = cost(S) + 1$ and $\kappa(\gamma) = cost(S)$ by the induction hypothesis, the claim holds.

- If an event was removed, then this event stems from the trace, and was not modified beforehand by assumption (†). Each alignment $\gamma' \in \Gamma(S')$ stems from some assignment μ' for C'. Ordering conditions and data conditions are independent. Thus there is an assignment μ that coincides with μ' on ordering constraints and assigns arbitrary data values compatible with C. Even though data values differ, for all moves except for the one concerning the removed event, their costs coincide, as required data values are enforced by dedicated conditions. The corresponding alignment $\gamma \in \Gamma(S)$ has a synchronous move where γ' has a log move, so $cost(\gamma') = cost(\gamma) + 1$. Since $cost(S') = cost(S) + 1$ and $cost(\gamma) = cost(S)$ by induction hypothesis, the claim holds.

- Suppose a data attribute v in an event $e = (\iota, a, \alpha)$ was freed. Each alignment $\gamma' \in \Gamma(S')$ stems from some assignment μ' for C'. Ordering conditions and data conditions are independent. Thus there is an assignment μ for C that coincides with μ' on ordering constraints and assigns data values compatible with C, in particular $\mu(v^\iota) = \alpha(v)$. Even though data values differ, for all moves except for the one concerning e, their costs coincide, as required data values are enforced by dedicated conditions. Since $cost(S') = cost(S) + 1$ and $cost(\gamma) = cost(S)$ by induction hypothesis, the claim holds.
- Suppose conditions were enforced. Then any model of C is also a model of C' as $C' \subseteq C$, and each alignment $\gamma \in \Gamma(S)$ that stems from some model μ for C is also an alignment in $\Gamma(S)$. Since $cost(S) = cost(S')$, the claim follows from the induction hypothesis. □

Let two alignments γ and γ' for $\mathbf{e}$ and $\mathcal{M}$ be *equivalent* if $|\gamma| = |\gamma'|$ and the move in γ at position i is a log/model/edit/synchronous move if so is the move in γ' at position i, and where edit moves edit the same variables. However, variable values in the model component of γ and γ' that do not match a variable in a trace may differ, as well as event identifiers.

Lemma B.3: Completeness

Let γ be an optimal alignment for $\mathbf{e}$ and $\mathcal{M}$. Then there is a goal state S_g in the search space such that $\Gamma(S_g)$ contains an alignment equivalent to γ.

Proof B.2: Lemma B.3

Let γ be an optimal alignment for $\mathbf{e} = \langle e_1, \ldots, e_n \rangle$ and $\mathcal{M}$. Let $\gamma|_M = \mathbf{f} = \langle f_0, \ldots, f_m \rangle$.

We show that there is a sequence of states $S_0, S_1, S_2, \ldots, S_g$ in the search tree such that, intuitively, by descending along this path the respective alignment gets closer and closer to γ, and S_g is a goal state that satisfies the claim. More precisely, we show that for each state $S_i = \langle E, C \rangle$, there is a correspondence relation $R_{S_i} \subseteq E \times \{f_0, \ldots, f_m\}$ associating some of its events with events in $\mathbf{f}$ that satisfies the following invariants:

(*i*) For every pair $(e, f) \in R_{S_i}$, the two events have the same activity.

(*ii*) For all events $e \in E \setminus \{e_1, \ldots, e_n\}$, there is some f_j in $\mathbf{f}$ such that $(e, f_j) \in R_{S_i}$. That is, all added events have a correspondent in $\mathbf{f}$. Moreover, for all edit or synchronous moves (e, f_j) in γ, f_j has a match in R_{S_i}.

(*iii*) Let $E_R \subseteq E$ be the set of events $\{e \in E \mid \exists f.(e, f) \in R_{S_i}\}$, μ the (partial) assignment on $vars(C)$ that sets, for all $(e, f) \in R_S$ with ι the id of e, $\mu(\tau^\iota) = time(f)$ and $\mu(x^\iota) = \alpha(x)$, where $f = (\iota', a, \alpha)$. Then μ satisfies $(C \setminus C_0)|_{V_R}$ where V_R is the union of all V^ι such that ι is an id of an event in E_R.

(*iv*) Only attributes of events are freed that have via R_S a correspondent in $\mathbf{f}$.

Moreover, the sequence of relations is monotonically increasing, i.e., we have $R_{S_0} \subseteq R_{S_1} \subseteq R_{S_2} \subseteq \ldots$.

We first show existence of a sequence $S_0, S_1, S_2, \ldots$ that satisfies the invariants, then we reason that it contains a goal state S_g that satisfies the claim.

At depth $k = 0$, we have $S = S_0$ and $E = \{e_1, \ldots, e_n\}$. Let R_{S_0} consist of all pairs (e_i, f_j) such that (e_i, f_j) is a synchronous or edit move in γ. The relation R_{S_0} satisfies (i) because in edit and synchronous moves the activity is shared. Item (ii) and (iii) are vacuously satisfied as no events were added, and (iv) because no attributes were freed.

Consider now a state $S = \langle E, C \rangle$ at depth k. If S is a goal state, we are done, so we assume that S has a violation. Suppose S gets repaired for a constraint $\psi = \langle \varphi, c_{act}, c_{tgt}, c_{cor} \rangle \in \mathcal{M}$. Consider first the case of a missing target violation; by a case distinction, we decide a next state S'.

- Suppose first that ψ does not have an activation. For simplicity, we consider the case of a single target, but the case for `Existence` is similar. Let f_j be the target of ψ in $\mathbf{f}$.
  - If there exists some $e = (\iota, a, \alpha) \in E$ such that $(e, f_j) \in R$ then e and f_j must have the same activity by (i), and we must have $C \not\models Ord(\psi, e) \wedge [c_{tgt}](e)$ $(\star)$, otherwise there would be no violation. Let $f_j = (\iota', a, \alpha')$.
    - Suppose there is some $v \in dom(\alpha)$ such that $\alpha(v) \neq \alpha'(v)$. This is only possible if e stems from the trace, since added events have no

fixed values. Thus, let S' be the child state obtained from a repair (5) where attribute v was freed.

* Otherwise, if $\bigwedge C \wedge Ord(\psi, e) \wedge [c_T](e)$ is satisfiable, let S' be the result of a condition enforcement repair (6). The repair is applicable because as observed above e and f_j have the same activity, and the resulting state is satisfiable because $C \wedge Ord(\psi, e) \wedge [c_{tgt}](e)$ is satisfiable.
* Otherwise, $\bigwedge C \wedge Ord(\psi, e) \wedge [c_{tgt}](e)$ is not satisfiable. Since μ satisfies $(C \setminus C_0)|_{V_R}$ and μ satisfies $Ord(\psi, e) \wedge [c_{tgt}](e)$, there must be some event $e' \in E$ from the trace, i.e. which adds conditions to C_0, that causes unsatisfiability. In fact, as assignments in e and f_j have no mismatches, $\bigwedge C \wedge Ord(\psi, e)$ must be unsatisfiable. Precisely, $Ord(\psi, e)$ must be $first(e)$ (resp. $last(e)$), but C contains $e' < e$ (resp. $e' > e$) for some trace event e'. Then e' cannot have a match in R_S because μ satisfies $C \setminus C_0$ restricted to $E|_{R_S}$ by (iii). Hence we can apply repair ((7)) to remove e', obtaining a state S'. Note that by invariant (iv), no attribute in e' can have been freed because it has no correspondent in R_S.

In all these cases R_S stays the same and thus satisfies conditions $(i) - -(iii)$. Moreover, in the first case the freed attribute belongs to an event that has a correspondent in R_S, so also (iv) is satisfied.

– Now assume there is no match for f_j in R. Then f_j cannot be in a synchronous or edit move in γ by invariant (ii). So f_j is in a model move, thus let S' be the state obtained from a repair (4) where a target event e was added. Set $R_{S'} = R_S \cup \{(e, f_j)\}$, which satisfies the invariants $(i) - (iii)$, and (iv) is satisfied because no additional attribute is freed.

• Now suppose ψ has an activation, let $e_{act} \in E$ be the activation event of the violation.

– Suppose there is some f_j such that $(e_{act}, f_j) \in R$. Suppose first that for $f_j = \langle \iota', a, \alpha' \rangle$, α' does not satisfy c_{act}.

* If there is some $v \in dom(\alpha)$ such that $\alpha(v) \neq \alpha'(v)$ then let S' be the child state obtained from a repair (2) where attribute v was freed.

* Otherwise, let S' be the result of a condition enforcement repair (3).

Second, suppose α' satisfies c_{act}, so it must have a target f_k in $\mathbf{f}$, $k \neq j$.

* If there exists some $e_{tgt} = (\nu, b, \beta) \in E$ such that $(e_{tgt}, f_k) \in R_S$ then we must have $C \not\models Ord(\psi, e_{act}, e_{tgt}) \wedge [c_{tgt} \wedge c_{cor}](e_{act}, e_{tgt})$ ($\star$), otherwise there would be no violation. Let $f_k = (\nu', b, \beta')$.
  - If there is some $v \in dom(\beta)$ such that $\beta(v) \neq \beta'(v)$ then e_{tgt} must stem from the trace. Let S' be the child state obtained from a repair (5) where attribute v was freed.
  - Otherwise, if $\bigwedge C \wedge Ord(\psi, e_{act}, e_{tgt}) \wedge [c_{tgt} \wedge c_{cor}](e_{act}, e_{tgt})$ is satisfiable, let S' be the result of a condition enforcement repair (6). The repair is applicable because of ($\star$), and as $\bigwedge C \wedge Ord(\psi, e_{act}, e_{tgt}) \wedge [c_{tgt} \wedge c_{cor}](e_{act}, e_{tgt})$ is satisfiable, and thus also the resulting state is satisfiable.
  - Otherwise, if $\bigwedge C \wedge Ord(\psi, e_{act}, e_{tgt}) \wedge [c_{tgt} \wedge c_{cor}](e_{act}, e_{tgt})$ is unsatisfiable, this must be because of some trace events in E that have no correspondent in $\mathbf{f}$ via R_S, since $\bigwedge(C \setminus C_0)|_{V_R} \wedge Ord(\psi, e_{act}, e_{tgt}) \wedge [c_{tgt} \wedge c_{cor}](e_{act}, e_{tgt})$ is satisfied by μ according to property (iii). In fact, as assignments in e and f_j have no mismatches (this was already excluded above), $\bigwedge C \wedge Ord(\psi, e_{act}, e_{tgt})$ must be unsatisfiable. Precisely, $Ord(\psi, e_{act}, e_{tgt})$ must be $e_{act} \ll e_{tgt}$ but C contains $e_{act} < e'$ and $e' < e_{tgt}$ (or similar for precedence) for some trace event e'. Then e' cannot not have a match in R_S, and apply repair (7) to remove e', obtaining a state S'. Note that by (iv) no attribute in e' has ever been freed because e' has no correspondent in R_S.
* Now assume there is no match for f_k in R_S. Then f_k cannot be in a synchronous or edit move in γ by property (ii). So f_k is in a model move. Let S' be the state obtained from a repair (4) where a target event e was added. Set $R_{S'} = R_S \cup \{(e_{tgt}, f_k)\}$, which satisfies the invariants.

– Suppose there is no $(e_{act}, f_j) \in R_S$. By invariant (ii), e_{act} stems from the trace. We then apply the repair (1) to remove the activation event

e_{act}, obtaining a state S'. Note that the activation event cannot have been modified, otherwise there would be a match in R_S by property (iv).

It can be checked that in all cases, the invariants $(i) - (iv)$ remain satisfied.

Second, consider an excessive target violation. Let e_{tgt} be the excessive target event that caused the violation.

- Suppose first that ψ does not have an activation. By a case distinction, we decide a next state S'. For simplicity, we consider the case of a single target, but the case for `Absence` is similar.
  - Suppose there is some f_j such that $(e_{tgt}, f_j) \in R_S$. Suppose first that for $f_j = \langle \iota', a, \alpha' \rangle$, α' does not satisfy c_{tgt}.
    - If there is some $v \in dom(\alpha)$ such that $\alpha(v) \neq \alpha'(v)$ then e_{tgt} must stem from the trace. Let S' be the child state obtained from a repair (9) where attribute v was freed.
    - Otherwise, let S' be the result of a condition enforcement repair (10).

    It can be checked that in all cases, the invariants $(i) - -(iv)$ remain satisfied.
  - Suppose there is no f_j such that $(e_{tgt}, f_j) \in R_S$. Then e_{tgt} must stem from the trace by property (ii). We apply repair (8) to remove an extra target.
- Now suppose ψ has an activation, let $e_{act} \in E$ be the activation event of the violation.
  - Suppose first there is some f_j such that $(e_{act}, f_j) \in R_S$.
    - Suppose there is some f_k in $\mathbf{f}$ such that $(e_{tgt}, f_k) \in R_S$. Let $e_{tgt} = (v, b, \beta)$ and $f_k = (v', b, \beta')$. Since $\mathbf{f}$ satisfies ψ, the assignment μ cannot satisfy $Ord(\psi, e_{act}, e_{tgt}) \wedge [c_{tgt} \wedge c_{cor}](e_{act}, e_{tgt})$.
      - If there is some $v \in dom(\beta)$ such that $\beta(v) \neq \beta'(v)$ then e_{tgt} must stem from the trace. Let S' be the child state obtained from a repair (9) where attribute v was freed.

· Otherwise, we apply a condition enforcement repair (10). If μ does not satisfy $Ord(\psi, e_{act}, e_{tgt})$ then we take the state $S' = \langle E, C' \rangle$ where $C' = C \cup \{\neg Ord(\psi, e_{act}, e_{tgt})\}$. Otherwise, μ does not satisfy $[c_{tgt} \wedge c_{cor}](e_{act}, e_{tgt})$, and we take the state $S'' = \langle E, C'' \rangle$ where $C' = C \cup \{\neg[c_{tgt} \wedge c_{cor}](e_{act}, e_{tgt})\}$.

∗ Suppose there is no f_k in $\mathbf{f}$ such that $(e_{tgt}, f_k) \in R_S$. Then e_{tgt} must stem from the trace. We apply repair (8) to remove the target event.

– Suppose there is no f_j such that $(e_{act}, f_j) \in R_S$. Then e_{act} must stem from the trace. We then apply repair (1) to remove an activation event.

It can be checked that in all cases, the invariants $(i) - (iv)$ remain satisfied.

This concludes the proof of existence of a sequence $S_0, S_1, S_2, \ldots$.

For $S = \langle E, C \rangle$ a state in this sequence, consider the measure $M(S) = (m - |R_S|, traceEvents(E), bnd(E), viol(S))$, where $|R_S|$ is the number of pairs in R_S, $traceEvents(E)$ is the number of events in E that stem from the trace, $bnd(E)$ the number of bound variables in events in E stemming from the trace, and $viol(S)$ the number of violations in S.

We observe that along the sequence $S_0, S_1, S_2, \ldots$, we have $M(S_0) > M(S_1) > M(S_2) > \ldots$, where we compare tuples lexicographically. Indeed, the measure decreases for repairs where an event was added because we always add an entry to R_S; for all other repairs, R_S stays the same. The measure decreases when removing an event as the number of trace events decreases; while for all other repairs the number of trace events stays the same. When freeing an attribute, the number of bound variables decreases, and when enforcing constraints, the number of violations decreases (while the number of bound variables is unaffected).

Thus by well-foundedness, we must at some point reach a state $S_g = (E, C)$ with $viol(S_g) = 0$, i.e., a goal state. We show that $m = |R_{S_g}|$: Suppose to the contrary that there is an event f_j in $\mathbf{f}$ with no e such that $(e, f_j) \in R_{S_g}$. There must be a move with f_j in γ, but it cannot be an edit or synchronous move by condition (ii). So it must be a model move. Since μ satisfies $C \setminus \{C_0\}$ restricted to $E|_{R_{S_g}}$, and S_g has no violation, so with Lemma B.1 we conclude that also $\langle f_1, \ldots, f_{j-1}, f_{j+1}, \ldots, f_m \rangle$ must satisfy

$\mathcal{M}$, which contradicts minimality of γ. Hence $m = |R_{S_g}|$, so every event in $\mathbf{f}$ has a matching event in E. By assumption (iii), assignment μ satisfies all constraints in C. Hence μ can be used in Alg. 4.1 to obtain an alignment of $\mathbf{e}$ and $\mathcal{M}$ that is equivalent to γ. □

Proof B.3: Correctness of the Data-Aware DECLARE Aligner (Theorem 4.1)

By Lemma B.2, every alignment extracted from a goal state is an alignment for $\mathbf{e}$ and $\mathcal{M}$. Moreover, by Lemma B.3, for an optimal alignment γ of $\mathbf{e}$ and $\mathcal{M}$ there is some goal state S_g that allows to extract an alignment equivalent to γ, and by Lemma B.2 it satisfies $cost(S) = \kappa(\gamma)$. The claim then follows from correctness of A*, i.e., the fact that a state with minimal cost is returned. □

This appendix also provides the complete search space for the running example introduced in Chapter 4, offering a detailed view of all states discovered during the exploration. It serves as supplementary material to the chapter, complementing the discussion and analysis presented therein. The full structure of the search space is visually represented in Figure B.1.

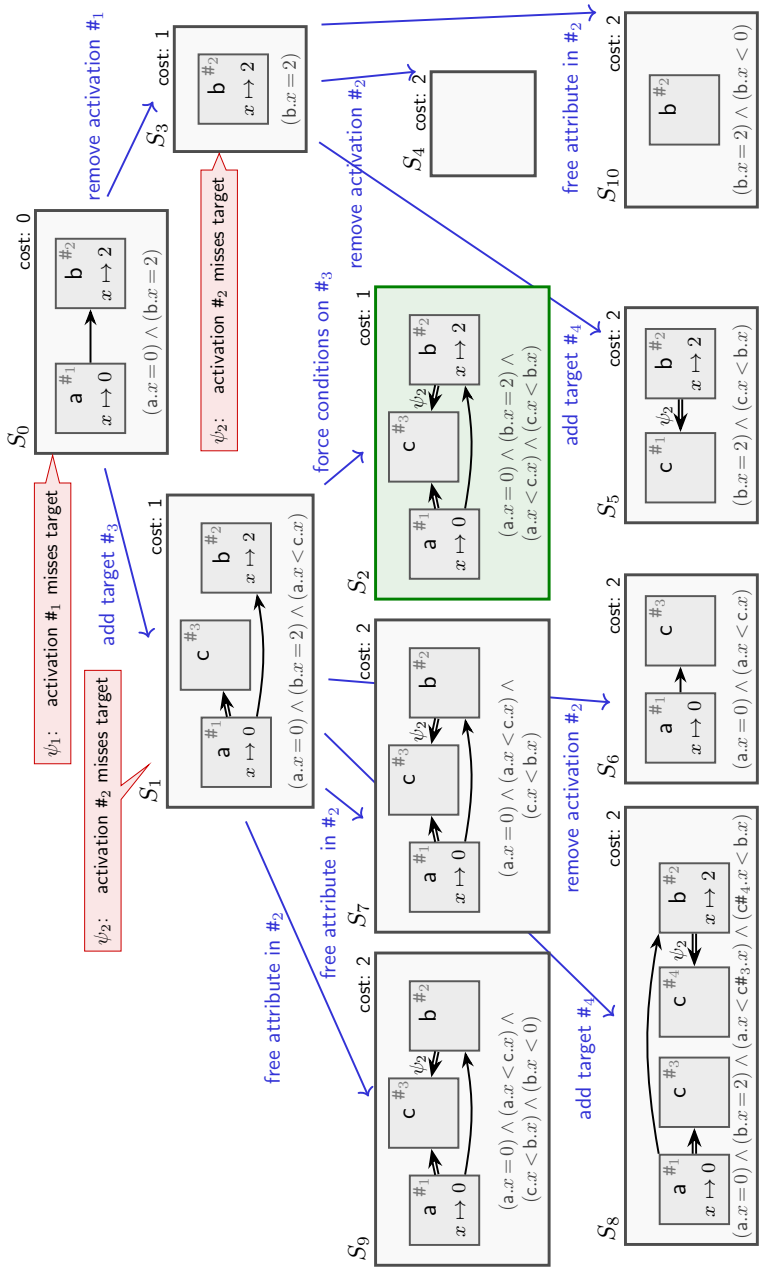


Figure B.1: Complete search space for the running example

Appendix C

Publications

This dissertation draws upon and partially reproduces material from several publications. The following sections provide a list of these publications, including a detailed description of the author's contributions to each work, as well as relevant copyright information.

C.1 Journals

Publication

Jacobo Casas-Ramos[a], Manuel Mucientes[a], and Manuel Lama[a]. REACH: Researching Efficient Alignment-based Conformance Checking. *Expert Systems with Applications*, 241:122467, 2024.
DOI: 10.1016/j.eswa.2023.122467

Journal Impact Factor (JCR 2024): 7.5

- Computer Science, Artificial Intelligence: **Q1** (28/204)

This paper is partially reproduced in Chapter 2. The PhD candidate contributed in the following CRediT [124] areas: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing - Original Draft, Visualization. According to the copyright terms stated by Elsevier:

[a]CiTIUS, Universidade de Santiago de Compostela, Santiago de Compostela, Spain.

Publication

Jacobo Casas-Ramos[a], Manuel Lama[a], and Manuel Mucientes[a]. DeclareAligner: A Leap Towards Efficient Optimal Alignments for Declarative Process Model Conformance Checking. *Engineering Applications of Artificial Intelligence*, 2025.
DOI: 10.1016/j.engappai.2025.111683

Journal Impact Factor (JCR 2024): 8.0

- Computer Science, Artificial Intelligence: **Q1** (25/204)

This paper is partially reproduced in Chapter 3. The PhD candidate contributed in the following CRediT [124] areas: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing - Original Draft, Writing - Review & Editing, Visualization.
According to the copyright terms stated by Elsevier:

[a]CiTIUS, Universidade de Santiago de Compostela, Santiago de Compostela, Spain.

C.2 International Conferences

Publication

Jacobo Casas-Ramos[a], Sarah Winkler[b], Alessandro Gianola[c], Marco Montali[b], Manuel Mucientes[a], and Manuel Lama[a]. Efficient conformance checking of rich data-aware declare specifications. In *Business Process Management*, pages 88–105, Cham, 2026. Springer Nature Switzerland.
DOI: 10.1007/978-3-032-02867-9_7

Conference ranking (ICORE, CORE2023): A

This paper is partially reproduced in Chapter 4. The PhD candidate contributed in the following CRediT [124] areas: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing - Original Draft, Writing - Review & Editing and Visualization.
According to the copyright terms stated by Springer Nature:

> d) The Licensee grants to Author the following non-exclusive rights to the Version of Record, provided that, when reproducing the Version of Record or extracts from it, the Author acknowledges and references first publication in the Volume according to current citation standards. As a minimum, the acknowledgement must state: "First published in [Volume, page number, year] by Springer Nature".
>
> i. [...]
>
> ii. [...]
>
> iii. to reuse the Version of Record or any part in a thesis written by the same Author, and to make a copy of that thesis available in a repository of the Author(s)' awarding academic institution, or other repository required by the awarding academic institution. An acknowledgement should be included in the citation: "Reproduced with permission from Springer Nature";

[a]CiTIUS, Universidade de Santiago de Compostela, Santiago de Compostela, Spain.
[b]Free University of Bozen-Bolzano, Bolzano, Italy.
[c]INESC-ID/Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal.

List of Tables

List of Figures

List of Definitions

List of Examples

This doctoral thesis has advanced the field of process mining by proposing novel approaches to optimal alignment-based conformance checking. The research has yielded a robust framework comprising methodologies that leverage heuristic search, pruning, and other optimization techniques to facilitate efficient conformance checking in procedural models. A new paradigm was also proposed for declarative models, which achieves high performance by focusing its actions on corrective measures that directly contribute to error rectification. Moreover, an approach has been proposed to accommodate expressive and flexible data conditions while preserving computational efficiency. These contributions have significantly enhanced the state of the art, resulting in more effective and accurate analysis of complex business processes.