"Quality-aware Automated Service Composition using Reverse Engineering and Incomplete Information"

Por

Ramide Augusto Sales Dantas

Tese de Doutorado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, MARÇO/2012

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RAMIDE AUGUSTO SALES DANTAS

“Quality-aware Automated Service Composition using Reverse Engineering and Incomplete Information"

ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR: Prof. Djamel Sadok
CO-ORIENTADOR: Prof. Carlos Kamienski

RECIFE, MARÇO/2012

Catalogação na fonte
Bibliotecária Jane Souto Maior, CRB4-571

Dantas, Ramide Augusto Sales
Quality-aware automated service composition using reverse engineering and incomplete information / Ramide Augusto Sales Dantas. - Recife: O Autor, 2012.
xi, 100 p. : il., fig., tab.

Orientador: Djamel Sadok.
Dissertação (mestrado) - Universidade Federal de Pernambuco. CIn, Ciência da Computação, 2012.
Inclui bibliografia.

1. Redes de computadores. 2. Composição de serviços. I. Sadok, Djamel (orientador). II. Título.

004.6 CDD (23. ed.) MEI2012 – 072

Tese de Doutorado apresentada por **Ramide Augusto Sales Dantas** à Pós- Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **"Quality-aware Automated Service Composition using Reverse Engineering and Incomplete Information"** orientada pelo Prof. **Djamel Fawzi Hadj Sadok** e aprovada pela Banca Examinadora formada pelos professores:

Visto e permitida a impressão.
Recife, 13 de março de 2012.

____

Prof. Nelson Souto Rosa
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Contents

Figure Index

Algorithm Index

Table Index

Abbreviations and Acronyms

ASC	Automated Service Composition
BPEL	Business Process Execution Language
CLM	Causal Link Matrix
DM	Dependency Matrix
EHC	Enforced Hill Climbing
FD	Fast Downward (planner)
FF	Fast Forward (planner)
GP	Graphplan (planner)
GUI	Graphical User Interface
HTN	Hierarchical Task Network
IPC	International Planning Competition
IT	Information Technology
OWL	Web Ontology Language
OWL-S	Web Ontology Language
PDDL	Planning Domain Definition Language
RDF	Resource Description Framework
RDFS	RDF Schema
SD	Service Developer
SOA	Service-Oriented Architecture
SOC	Service-Oriented Computed
STRIPS	Stanford Research Institute Problem Solver
SWRL	Semantic Web Rule Language
WS	Web Services
WSDL	Web Services Description Language
WSML	Web Service Markup Language

Abstract

Service Composition is one of the most important features offered by Service Oriented Computing. The composition allows a new service to be created through the reuse of existing ones. The process of creating a composition involves discovering the necessary services and combining them in an appropriate manner using specific languages and tools. This process, however, is still carried out mainly by hand. Considering the dynamic nature of distributed services, manual composition may become too complex, affecting the productivity gains provided by reuse. Proposals to fully or partially automate this process already exist, most of them based on Automated Planning algorithms borrowed from Artificial Intelligence. Although functional, these approaches have practical problems that hinder their effective implementation in production scenarios. In this Thesis, we addressed some of the practical problems of automated composition, starting with the need for formal descriptions of services. These formal descriptions are necessary for the composition algorithms, however, are rarely available from services. This issue was addressed by means of reverse engineering a repository of service compositions. By analyzing how the services were related to each other in the compositions, it was possible to obtain the necessary information for the algorithms to work. We also evaluated the quality of the compositions generated by the algorithms and their similarity with respect to compositions created manually. Automated Planning algorithms from the literature have been modified in order to generate solutions closer to those expected by the developer. Finally, the composition algorithms were adapted to accept incomplete specifications, thus allowing the developer to obtain a solution even not knowing a priori all the composition details. Comparisons with automated planning tools were conducted in order to ascertain the effectiveness of the algorithms. The results show that the automated composition, as presented in the Thesis, can be an invaluable tool to the service developer.

Keywords: Service Composition, Web Services, Automated Planning.

Resumo

A composição de serviços é um dos recursos mais importantes disponibilizados pela Computação Orientada a Serviços. A composição permite que um novo serviço seja criado através do reuso de serviços existentes. O processo de composição envolve a descoberta dos serviços relevantes e sua combinação de forma adequada, por meio de ferramentas e linguagens específicas. Esse processo, no entanto, ainda é realizado essencialmente de forma manual. Considerando a natureza distribuída e dinâmica dos serviços, a composição manual pode se tornar demasiado complexa, comprometendo os ganhos em produtividade proporcionados pelo reuso. Propostas para automatizar completa ou parcialmente esse processo já existem, a maioria dessas propostas é baseada em algoritmos de planejamento automatizado emprestados da Inteligência Artificial. Apesar de funcionais, essas abordagens possuem problemas práticos que dificultam sua aplicação efetiva. Nesta Tese foram abordados problemas práticos da composição automatizada, começando pela necessidade de descrições formais dos serviços. Essas descrições formais são necessárias aos algoritmos de composição, porém, raramente estão disponíveis junto aos serviços. Esse problema foi abordado por meio da engenharia reversa de um repositório de composições de serviços. Analisando como os serviços estavam relacionados entre si nas composições, foi possível obter as informações necessárias aos algoritmos de composição. Também foi avaliada a qualidade das composições geradas pelos algoritmos quanto a sua semelhança com relação a composições criadas manualmente. Algoritmos da literatura de planejamento automatizado foram modificados com o objetivo de gerar soluções mais próximas das esperadas pelo desenvolvedor. Finalmente, os algoritmos de composição foram adaptados de forma a aceitar especificações incompletas, assim permitindo ao desenvolvedor obter uma solução mesmo não sabendo *a priori* todos os seus detalhes. Comparações com ferramentas de planejamento automatizado foram conduzidas com o objetivo de averiguar a efetividade dos algoritmos. Os resultados mostram que a composição automatizada, como apresentada nessa Tese, pode ser uma ferramenta valiosa ao desenvolvedor de serviços.

Palavras-chave: Composição de Serviços, Web Services, Planejamento Automatizado.

Chapter 1 Introduction

1.1. Motivation

The Internet is increasingly becoming a service – not only a content – network, with various organizations using its tools and protocols to provide and access services from other parties and leverage their businesses. Amazon, the book retailer and cloud provider, for example, has just recently launched[1] the Simple Workflow Service (SWF) [3]. The SWF service allows organizations to host and maintain their businesses processes and services using the Amazon's cloud infrastructure. The OASIS organization defines service as "a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description" [42]. The use of services as building blocks for distributed applications has led to the emergency of the Service Oriented Computing (SOC) paradigm [12], which has been part of corporate IT strategies for years already. In this context, the Web Service technology [9] has played an important role, enabling enterprises and Web sites to offer, locate and access services worldwide using standard Web protocols.

Today's Internet offers a multitude of Web-based services to service developers and advanced users. With services varying from access to stock prices and weather forecast to advanced enterprise-wide services, the Web has become a playground for innovative and ambitious applications developers. Those who can create and make available new services quickly stand better chances to leap ahead of competitors and succeed. The same way standalone software is built using known libraries of functions or classes, a new service can also descend from more general, "smaller" services that are composed together to provide the new functionality. The arrangement of these services can be made using regular programming languages (Java, C#, etc.) or languages specially tailored for this purpose (for example, WS-BPEL [44]).

[1] Amazon SWF launched in February, 2012.

This task, however, has become complex and time consuming. The increasing number of services available today imposes a challenge to service developers, and the lack of organization of these services adds up to the problem. Unlike application APIs in traditional programming languages, Web Services are not sorted into libraries or packages, thus requiring a developer to search for the desired functionality within a wider range of options, with different descriptions and interfaces. Moreover, during the process of combining these services, new interesting ones may come up and services already in the composition can change or be removed, leading to frequent changes in the service process.

The difficulty in creating and maintaining new services out of existing ones has led Academy and Industry to pursue new ways for a smoother, yet powerful, services development process. The idea is to streamline service creation in order to keep pace with the dynamicity of technology markets. In one foreseeable scenario, users with low technical skills would be able to create and make available their own services – for profiting from them or simply to share with friends on Facebook for example. Examples like those from Facebook, Apple and Firefox – which provide rich platforms for application developers, leveraging their own products – can inspire other IT providers. Telecommunication Operators that no longer want to be simple data "pipes" should also invest to provide more flexible (and profitable) service platforms.

The notion of Services can be extrapolated to infrastructure services and not be limited to the "business" level. Services can be used to control telecom and IT infrastructure "services" (e.g., data paths, telephony intelligent services, messaging), thus making business and technology work under a common design paradigm, that of services creation and their composition. Initiatives as the Parlay X [51] have proposed WS interfaces to common telecom services, so that application developers could take advantage of them to address new business opportunities. One also can see a future where devices and equipments would leave the factory floor with WS-compatible APIs and hence be ready for use by composition techniques to deploy new functionalities to the network. In this sense Web Services can be a powerful replacement to MIB/SNMP-based management [43]. This kind of solution also fits nicely into the Platform and Infrastructure as a Service ideas (PaaS and IaaS, [65]), offering standard-compliant interfaces to platform management software.

The best scenario for the efficient service creation would be to completely automate the act of composing services, but it brings new challenges by itself. Several approaches have been proposed to address automating service composition (described in Chapter 3), which try to completely automate the process of finding and combining services to achieve a user-specified functionality. These approaches are often inserted in the context of the Semantic Web (SW) [6], comprising the more specific Semantic Web Services research field [36]. The Semantic Web advocates the idea of semantic-equipped Web content over which automated agents could reason and generate new content; Semantic Web Services extend this idea to more "active" content, i.e., services.

In the fully automated Service Composition solutions, Artificial Intelligence (AI) planning is a recurring technique and presents interesting results. Using logic-enabled semantic languages provided by the SW community (for example, DAML-S [3], OWL-S [70] and WSML [31]), automated composition tools can find service compositions meeting user requests provided that the underlying services are completely and correctly described in terms of their capabilities. These approaches go beyond typical WS description – i.e., WSDL – which focuses on the service interface by looking at enriched Input and Output descriptions (e.g., [78], [4], [32]), pre- and postconditions (e.g., [1] and [2]) and, eventually, even internal execution flows [37]. Several factors, however, prevent the adoption of such proposals by the industry. In this Thesis, we identified and addressed four of these factors, as listed below:

- **Services need to be formally described in order for algorithms to work properly.** The process of describing the services is manual, time-consuming and error-prone, requiring special language skills different from the usual skills of service developers. That explains why despite semantic description languages being available for years already, semantic-described services (in production) are still very uncommon.
- **Algorithms efficacy and efficiency.** The problem of finding a set of services among several candidates that can be combined in the correct order to provide a given functionally is widely accepted as a complex undertaking [5], [20]. The existing approaches tackle the problem by limiting the complexity of the solutions (e.g., not allowing control structures such as if's and loops), thus reducing the search space. Nonetheless, by limiting the complexity of solutions, a whole set of classes of compositions can no longer be generated, hence reducing the applicability of such approach. The alternative is hardly more attractive: a complete search for full-fledged service compositions over the space of thousands of services

may not terminate at all. A compromise need to be found between expressiveness and efficiency in order for a practical solution to be possible.

- **Lack of confidence in the resulting compositions.** Automated composition algorithms can render correct-by-construction service compositions provided that the domain is correctly described (what is not simple, given point 1). However, developers and service providers are not willing to deploy new services without knowing whether they meet users' expectations and do not risk compromising their infrastructures. It is expected that the developer will have to deal with the result of the automated process, fixing and adapting the composition to ensure that it achieves its purpose. The quality of the automated solutions will determine how much extra effort will be needed by this task.
- **Lack of adequate tools for service specification.** Providing the initial service specification as an input to a composition algorithm is not a straightforward task. Detailed knowledge in the underlying formalisms used by algorithms is often required so that meaningful and sound specifications can be written. Automated solutions are often little tolerant to malformed or incomplete specifications, and this all-or-nothing behavior ends up harming the developer's productivity.

1.2. OBJECTIVES

The general objective of the Thesis is to facilitate the creation of new services through the process of service compositions, focusing on Web Services for the matter of practicality, but not being limited to WS technologies. In particular, in this Thesis we addressed the scenario with Automated Service Composition (ASC), which we believe can help to improve the productivity of developers and reduce the time-to-market of new services. We addressed the issues with ASC listed in the previous sections as described below:

- **Need for semantic descriptions.** Instead of working with manually written descriptions of services using Semantic Web languages, we opted to apply a Reverse Engineering approach in order to obtain the semantic information needed by automated algorithms. We used compositions available at the myExperiment project [60], a repository of scientific workflows and social networks that promotes the exchange of data-intensive service compositions.
- **Algorithms efficiency and efficacy.** We chose to work with existing algorithms from Automated Planning literature that showed suitable to the problem of Automated Service Composition. We simplified the algorithms, in order to keep

them efficient, at the same time that we extended them to improve the quality of the solutions.

- **Confidence on automated solutions.** In order to establish the quality of the compositions generated via ASC we conducted a performance evaluation where our algorithms faced an award-winning AI planner. The results showed the overall quality of the solutions is acceptable using the AI planner but can be improved for specific scenarios – e.g., when not all input parameters were used – using our algorithms.
- **Composition specification.** In early works, we worked on a Graphical User Interface that allowed a composition to be specified in a mixed way: part manually and other parts left to the ASC algorithm to complete. In this Thesis, we addressed the related problem of incomplete service specifications, where the developer does not provide enough information for an exact solution to be found but expects from the composition algorithm at least an approximate response.

In special, in this Thesis we focused on Automated Service Composition as an auxiliary tool for the composition developer, not as a complete solution that will take his/her place in the development cycle. Although this latter case seems to be the target scenario for most ASC proposals, we believe that there is a long way before complete automation of service creation can be achieved.

1.3. ORGANIZATION OF THE THESIS

This Thesis is organized as follows. Chapter 2 brings some background on the Service Oriented paradigm and Automated Planning. Chapter 3 describes current service composition techniques and discusses their strengths and limitations. Chapter 4 presents the reverse engineering process employed to extract semantic information from repository of scientific workflows myExperiment. In Chapter 5 we describe our extensions to planning algorithms aimed at increasing the quality of compositions. A performance evaluation of the algorithms and comparison to a state-of-the-art planner is presented in Chapter 6. Chapter 7 concludes this Thesis with the contributions, future work and final remarks.

Chapter 2 Background

2.1. Service Oriented Computing

Service Oriented Computing is a paradigm that employs "services" as the basic unit for the separation of concerns in computing systems' development [20]. Services can be seen as the means by which consumers access providers' capabilities [63]. Among other interesting features, services provide loosely coupled interaction between partners in a business process (or any other computing activity). Partners can implement services in diverse ways and still cooperate as long as clear interfaces are made available.

Service-oriented architectures (SOA) have been dictating distributed systems construction in the last years, especially across organizational boundaries. SOA is a paradigm for the organization, design, implementation and utilization of distributed capabilities packaged as services [20]. Enterprises see in SOA an opportunity to boost Business-to-Business (B2B) transactions and reduce integration costs of legacy systems. In the latter case, old systems are equipped with service-oriented APIs, which enable them to interact with new ones. To achieve such interoperability, well-known protocols and languages are necessary. The most common realization of service-oriented architectures is through the use of Web Services (WS).

Web Services are an XML-based technology that has as main protocols and specifications SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language) and UDDI (Universal Description, Discovery and Integration) [16]. For network transport capabilities, Web Services utilize standard Internet protocols: HTTP, SMTP, FTP, among others. Web Services messages are built and exchanged at the application layer using SOAP for XML serialization of method invocations and objects. WSDL is used to describe web services interfaces, i.e., how other web services can access it; WSDL service descriptions can be published and located through the UDDI directory service. The stack of protocols used by Web Services is shown in Figure 2-1.

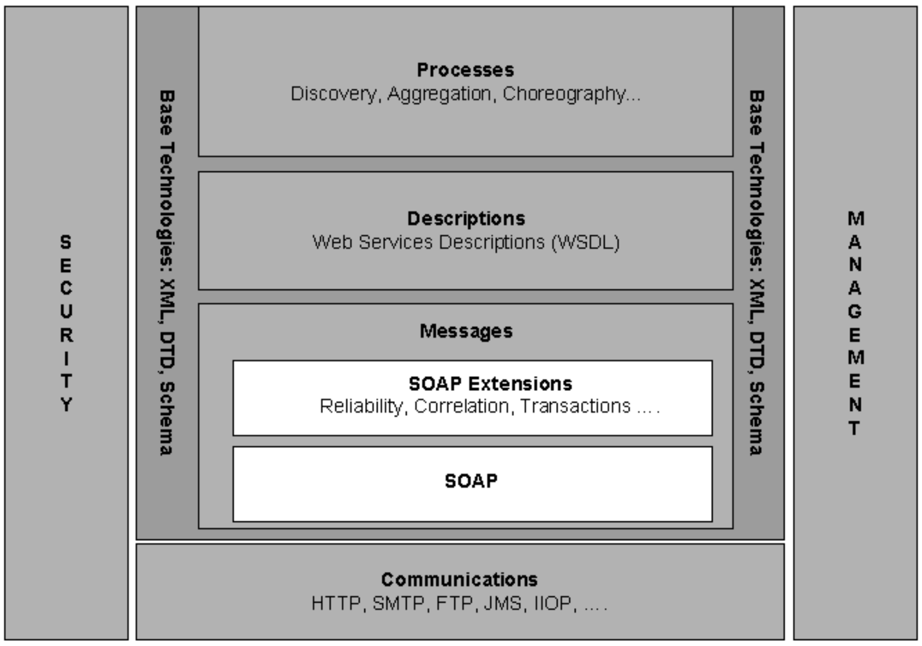

SECURITY
Base Technologies: XML, DTD, Schema
Processes
Discovery, Aggregation, Choreography...
Descriptions
Web Services Descriptions (WSDL)
Messages
SOAP Extensions
Reliability, Correlation, Transactions ....
SOAP
Base Technologies: XML, DTD, Schema
MANAGEMENT
Communications
HTTP, SMTP, FTP, JMS, IIOP, ...

Figure 2-1. Web Services protocol stack [27].

Web Services take advantage of the ubiquitously deployed Web protocols and standards to allow XML-encapsulated remote service invocation. Organizations can publicize internal services as Web Services and allow other organizations to find and use them. Intra-organization tasks can also benefit from Web Services high interoperability. The most evident drawback of using the Web stack along with XML parsing/encoding is the built-in overhead, what makes service invocation suboptimal compared to pure RPC-like technologies. Nevertheless, as computer power increases and even small, low-capacity devices become able to run web servers (e.g., DSL modems); such overhead can compensate the gain in interoperability. Another drawback comes from the effort needed to change ones development paradigm within companies with long established development traditions and a large set of accumulated "legacy" software. It is hoped that the benefits of Web Services will outweigh such limitations and soon become ubiquitous.

More studies point the benefits of using SOA [20], [77]. It is believed to facilitate the growth of large-scale enterprise systems, allow Internet-scale provisioning and service usage and reduce costs in the cooperation between organizations. The main value of SOA is that it provides a simple scalable paradigm for organizing large networks of systems that require interoperability to draw from the value in the individual components. SOA scalability derives from its almost-null assumptions about the underlying network and trust that are often implicitly made in small-scale systems.

2.1.1. Service Composition

Composing services into business processes is a fundamental, yet potentially complex task in service-oriented design. A service composition is a coordinated aggregation of services that have been assembled to provide the functionality required to automate a specific business task or process [20]. Services are expected to be capable of participating as effective composition members, regardless of whether they need to take part in a composition from scratch. As the sophistication of service-oriented solutions continues to grow, so does the complexity of underlying service composition configurations.

Service Orchestration and Choreography are two concepts used to tackle service composition complexity, see Figure 2-2. Both regard to the coordination or control of individual services in order to make them work together in a coherent manner [70]. Orchestration refers to coordination at the level of a single participant's process, specifying the control and data flows needed for the process to be executed correctly. Among the several existing languages to describe service orchestration, the most representative is Business Process Execution Language for Web Services (WS-BPEL, or just BPEL) [12], [76], [65]. Service Choreography refers to the protocol that ensures harmony and interoperability among the interacting participants (processes), in order for the processes to cooperate with no dependency on a centralized controller. Choreographies specify the temporal relationship in message exchanging, providing rules to the correct execution of the individual processes. The most accepted language for service choreography is W3C's Web Service Choreography Description Language (WS-CDL) [86].

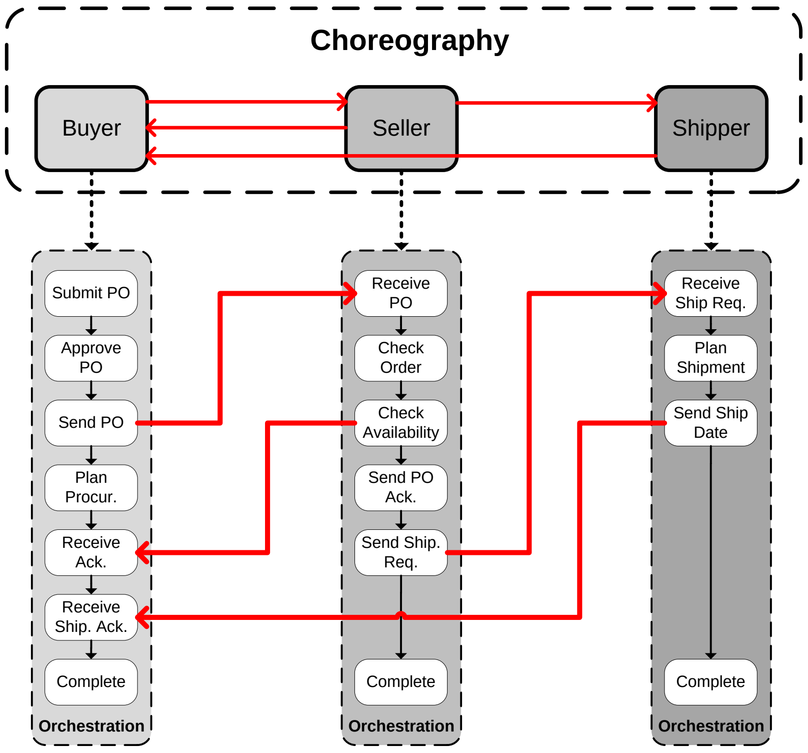


Figure 2-2. Example of Service Orchestration and Choreography [43].

An example of the relation between service orchestration and choreography is shown in Figure 2-2, reproduced from [43]. This picture illustrates a classical example involving an internal process for buyers (procurement), another process for sellers, and yet a third one for shippers. Each one of these trading partners will be executing its own internal process. Procurement, for instance, could involve approvals at multiple layers. These business processes are the orchestration of activities within each partner. Choreography, on the other hand, does not focus on the details of the internal orchestration process; rather, the focus is on the information exchange among the participants.

2.2. Semantic Web

The idea of a Semantic Web [9], term coined by Web's inventor Tim Berners-Lee, views the Web as a global knowledge base that machines could interpret and operate over. To accomplish this perspective, efforts were directed to find ways of enhancing Web content with meta-information carrying a formally specified semantic description of the content being published. This would allow not only humans but also software agents to understand and aggregate content from several sources, filter it and generate new, valuable knowledge to users or other computer agents.

Ontologies have been adopted as the preferred way to describe content for the Semantic Web. The following sections describe the concept of Ontology, present languages and tools used for semantic descriptions.

2.2.1. Ontologies

Ontology is a concept borrowed from Philosophy for which there are several definitions [29], [93], [85] and [91]. One definition describes ontology as an explicit and formal specification of a shared conceptualization [28]. It is explicit because it describes unambiguously concepts, their properties, relationships, functions, axioms and restrictions – allowing it to be read and interpreted by machines –, and it is a conceptualization because it is an abstract model and a simplified view of the entities it represents. Finally, it is shared because it has been agreed upon by a group of experts. To sum up, an ontology is the definition of a set of concepts, their taxonomy, interrelation and the rules that govern these concepts [85].

Ontologies represent knowledge in a similar way Object Oriented approaches do. There are means for defining static structures such as classes (concepts), class hierarchies (using inheritance), attributes, relationships and instances. The significant differences are within the representations of implicit semantics often left hidden in Object Oriented models. Ontologies allow the use of constraints, axioms and rules whereas object-orientation relies on methods (sequences of imperative commands) associated with classes and objects for semantic verification. Both approaches are equally expressive although a declarative approach is better suited for a domain modeling than an imperative one [94].

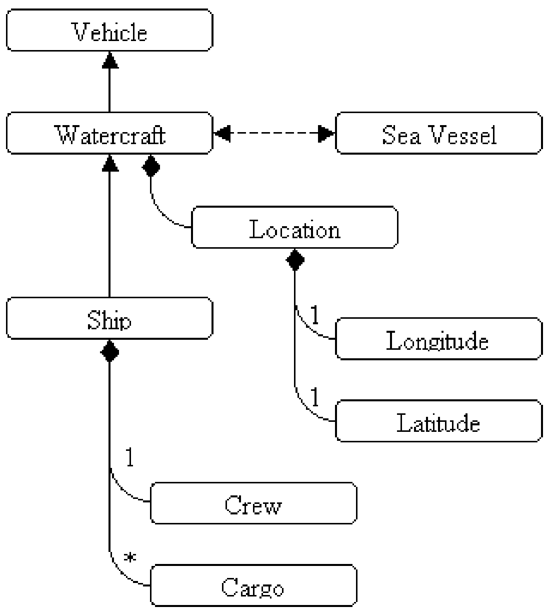


Figure 2-3. Example of Ontology.

A simple ontology for the domain of vehicles is graphically represented in the Figure 2-3 above. In this example, Ships are a kind of Watercraft or Sea Vessel. Ships have a Crew and Cargo. Through the transitivity of the hypernym relation, Ships also have a Location. The Location of a Ship consists of a Longitude and Latitude. Vehicle, Sea Vessel, Watercraft and Ship are classes; Location, Crew, Cargo, Longitude and Latitude are Slots or Properties.

2.2.2. Ontology Languages

In computer science and artificial intelligence, ontology languages are used to encode knowledge about specific domains and often include reasoning rules to allow the processing of this knowledge. Foundational ontologies (sometimes called “upper level ontologies”) define a range of top-level domain-independent ontological categories, which form a general foundation for more elaborated domain-specific ontologies (e.g., the Unified Foundational Ontology (UFO) [30]). Foundational ontologies can be used to evaluate conceptual modeling languages and to develop guidelines for their use.

Several ontology languages have been created in the semantic Web community, usually based on frame-based representation of knowledge [58]. Frame languages are rather focused on the recognition and description of objects and classes; relations and interactions are not given the same importance. Generally speaking, “frame” in this context means "something that can be fulfilled". For instance, object oriented languages are frame-based languages. Examples of traditional ontology languages are Frame Logic [44], the

Knowledge Interchange Format (KIF) [47], Ontolingua [21], LOOM [54], the Operational Conceptual Modeling Language (OCML)[59], and the Open Knowledge-Base Connectivity (OKBC) [73]. Frame Logic (F-Logic) [44] is seen as the most prominent among them. It integrates frame-based languages and first-order predicate calculus. It covers in a clean and declarative manner most of the structural aspects of object-oriented and frame-based languages, such as object identity, complex objects, inheritance, polymorphic types, query methods and encapsulation. F-Logic has a model-theoretic semantics and a sound and complete resolution-based proof theory [15]. Applications of F-Logic go from object-oriented and deductive databases to ontologies, and it can be combined with other specialized logics to improve the reasoning with information in the ontologies.

As Semantic Web concepts are developed and disseminated, Web-born knowledge representation languages gather attention. W3C-endorsed languages such as XML, RDF (Resource Description Framework) and RDFS (RDF Schema) have been used for equipping Web content with meta-data information for machine recognition. RDF aims at providing a mechanism for describing resources making no assumption about a particular application domain nor the structure of the document being described. RDF Schema (RDFS) is a declarative language used for the definition of RDF schemas. The RDFS data model (which is based on frames) provides mechanisms for defining the relationships between properties (attributes) and resources.

Web Ontology Language (OWL) [88] is a family of knowledge representation languages for authoring ontologies – also endorsed by the W3C –, whose necessity arose from a review of the DAML+OIL language [17]. OWL improves machine interpretability of Web content compared to RDF and RDF Schema by providing additional vocabulary and formal semantics, being usually serialized using RDF/XML syntax. OWL is considered one of the fundamental technologies underpinning the Semantic Web, and has attracted both academic and commercial interest. OWL allows developers to describe the relationships between the system's classes and individuals via axioms, from which new relationships can be inferred. This family of languages is comprised of three increasingly expressive sublanguages: OWL Lite, OWL DL, and OWL Full. OWL DL and OWL Lite semantics are based on Description Logics [37], which have attractive and well-understood computational properties; OWL Full uses a novel semantic model intended to provide compatibility with RDF Schema. Yet, OWL constraints are not powerful enough to govern behavior of many different-class entities together.

The Semantic Web Rule Language (SWRL) [87] is a proposal for a Semantic Web rules language, combining sublanguages of the OWL Web Ontology Language (OWL DL and Lite) with those of the Rule Markup Language [82]. SWRL includes a high-level abstract syntax for Horn-like [36] rules in both the OWL DL and OWL Lite sublanguages of OWL. SWRL gives more power to the ontology developer once it permits to deal with more than one class in a rule. That way SWRL rules can reach a broader range of actions and define ontological constraints and complex behaviors more properly than pure OWL.

The OWL-S ontology [89] is an extension of OWL for describing Services. An OWL-S description of a service comprises three parts: Service Profile, Process Model and Grounding. The Service Profile provides a concise description of the service for the purpose of service discovery by the developers. Besides basic information about the service, like name and textual description, the Profile brings information relevant for its usage by the clients: its inputs and output parameters along with the preconditions and results (effects) of the service. The Process Model, on the other hand, provides an internal view of the service by detailing how the clients should interact with it. It is comprised of Atomic Processes descriptions, for simple, usually stateless request-response services, and Composite Processes, used to describe multi-staged, stateful services. The former is akin to current WSDL-described services, while the latter models services that interact with clients in a sophisticated way, with support to choices, loops, parallelism, etc. Finally, the Grounding part details how to access the service, complementing its WSDL description. It provides the mapping between OWL-S processes and WSDL operations and the format of the messages exchanged with the service.

2.3. AUTOMATED PLANNING

Automated Planning is the discipline of Artificial Intelligence concerned with the elaboration of action plans for automated agents. Informally, a planning task is comprised of problem and domain descriptions. The planning domain describes the elements that comprise the domain – e.g., variables, constants and their relations – and the actions applicable to the domain. Actions are described in terms of their observable behavior, i.e., the effects they have over the domain elements. Each action has a set of preconditions – which must hold in order for it to be applicable at a given moment – and a series of effects that change the state of the system. A planning problem defines the initial state the system – i.e., what is assumed as being true before the plan executes – and the set of acceptable states (goal states) the system must be after the execution of the plan. Plans are comprised of actions, usually ordered, although a total order is not always required. Valid plans are those where, for all actions in the plan, the preconditions hold if the order of execution of the plan is followed. A plan is a solution for a planning problem if it is valid and accepts the initial state of the problem (the plan's precondition holds in the initial state) and the state brought by executing the plan is a valid goal state of the problem. The content of this section was adapted from [25].

2.4. Classical Planning

In this Thesis, we used the more restricted but frequently used Classical Planning model, along with some extensions. Classical Planning is often referred to as STRIPS planning after the Stanford planning system and language that laid the ground for several planners since its proposition [22]. Classical Planning can be seen as planning for a transition system subject to a series of restrictions:

1. The system has a finite set of states;
2. The states of the system are fully observable, i.e., one has complete knowledge about any given state;
3. The system is deterministic, i.e., for any given state, applying an action leads to one and only one next state.
4. The system is static, i.e., it changes only by the application of some action; no other external events change its current state.
5. The system supports only restricted goals, i.e., goals that describe the final state (or set of final states) of the system – extended goals, on the other hand, would allow the definition of intermediate states to be traversed by the plan;
6. The solutions for a planning problem is a linearly ordered sequence of actions;
7. The system has implicit time, meaning that actions have no durations associated to them;
8. The planning process is performed offline, unaware of any dynamics that affects the state of the system.

In the algorithms we developed, we relaxed restrictions 5 (restricted goals) and 6 (plans as sequences of actions). The latter restriction was relaxed due to the option of working with Planning Graph algorithms, which natively generate plans that consist of sequence of sets of actions (actions in each set are independent and can be executed irrespective of order). We relaxed the restriction 5 in the process of enhancing our algorithms to increase the quality of the plans generated. These changes are explained in Chapter 5, where we describe the algorithms in detail.

Formally, a planning domains and problems can be represented using a Set-theoretic approach. Let $\Sigma = \langle \mathrm{S}, \mathrm{A}, \gamma \rangle$ be a restricted transition system as described in the previous section, where S is the set of states the system can assume, A is the set of actions, and $\gamma: S \times A \rightarrow S$ is the state transition function. In the Set-theoretic approach, a state is given by the set of proposition literals that are true at that state. This representation uses the Closed World Assumption, i.e., propositions not present in the state are assumed to be false. An action in this representation is comprised of three sets of literals: the set of preconditions, $precond$, the action positive effects, $effects^{+}$, and its negative effects $effects^{-}$, with sets $effects^{+}$ and $effects^{-}$ disjoint. The action $a = \langle precond(a), effects^{+}(a), effects^{-}(a) \rangle$, for example, is applicable to state $s = \{l_1, l_2, \dots, L_n\}$, where l_i, $i = 1,2,\dots,n$ are the literals with truth-value $TRUE$ in state s, if and only if $precond(a) \subseteq s$. The resulting state of applying a to s, i.e., $\gamma(s,a) = s'$, is:

$$s' = \left(s - effects^{-}(a)\right) \cup effects^{+}(a)$$

A planning problem $\mathcal{P}$ in the set-theoretic representation is a triple $\mathcal{P} = \langle \Sigma, s_0, g \rangle$, where Σ is the restricted transition system, s_0 is the initial state and g a set of propositions that define the goal states. A solution to problem $\mathcal{P}$ is a plan that takes the system from the initial state s_0 to any state s such that $g \subseteq s$.

An alternative way of representing planning domains and problems, called Classical Representation, uses predicates over domain variables instead of propositions. Assume for example a domain where robots move around numbered locations. Instead of using proposition literals such as $atA1$ – meaning that robot A is at positions 1 –, a predicate $at(R_x, L_n)$ is defined, where R_x stands for some robot and L_n a location, and used for any combination of robots and locations of the domain (e.g., $at(R_a, L_1)$). Action preconditions and effects become first-order formulas over these predicates in the Classical Representation. The truth-value of these predicates can change depending on the state – robot R_a moves to location L_2 and $at(R_a, L_1)$ evaluates to false now –, and therefore they are often called *fluents* or flexible relations. Both classic and set-theoretic representations have the same expressive power in the sense that a planning domain described using the classical representation can also be equally represented using set-theoretic concepts. The planning language PDDL [24], explained in Section 2.4.2, uses a notation compatible with the Classical Representation.

2.4.1. Planning Techniques

Planning problems can be solved using a multitude of techniques, which can be grouped according to the way they approach the planning problems. In this section, we provide a glimpse of the existing approaches.

- **State-space techniques** address the planning problems in a straightforward way: searching the state space from the initial state for a sequence of states that lead to any of the acceptable goal states, keeping track meanwhile of the actions that caused state changes. This direct approach is called Forward Search (since it starts from the initial state) but Backward State-space search algorithms exist – such as the original STRIPS algorithm – which starts from the goals and try to reach s_0. The sequence of actions found will comprise the resulting plan. Obviously, if the state space is large the search can take a considerable time to be complete (but under Classical Planning, it will eventually finish since the state space is finite). That fact led to the development of heuristics for searching the states space more efficiently but sacrificing completeness – STRIPS itself is an incomplete heuristic.

- **Plan-space techniques** approach the problem from a different perspective. Instead of states, the search operates over a set of partial plans. It starts with an initial plan and applies successive "fixes" – each fix results in a new partial plan in the plan space – until a valid plan is found. Fixes include adding a new action so that the precondition of another action already in the plan is provided, changing the order of actions in order to prevent conflicts (e.g., the effect of one action negating the precondition of another), etc. The solution found by a plan-space algorithm is typically a partially ordered set of actions.

- Other techniques employ a structure called **Planning Graph** to model the planning problems. The Planning Graph is a layered graph, each level comprised of propositions and the actions whose preconditions hold at this level (i.e., the preconditions are contained in the level propositions). The next level propositions are given by the (positive) effects of the actions at the level above. The top-level propositions are given by the problem's initial state, and the graph is expanded until a level whose proposition set contain the goal state is reached. Once a "goal level" is found, a backward search is performed in the graph in order to extract a plan, which is given as a sequence of sets of actions. A more detailed description of this procedure can be found in Chapter 5.

- **Propositional Satisfiability Planning techniques** try to take advantage of the extensive research in algorithms for solving propositional satisfiability problems to leverage automated planning. The general approach consists of encoding the planning problem into a satisfiability problem – i.e., a Boolean expression with only conjunctions and disjunctions of propositions – and submitting it to SAT solvers. In case the solver finds a model – i.e., an assignment of truth values to proposition that render the formulation true –, it is translated back to a plan.

Several other approaches for solving planning problems exist, for a complete reference see [25]. In this Thesis, we are interested in Planning Graph-based techniques as well as State-space search, which present interesting properties given the way we modeled the composition problem (details in the Chapter 5).

2.4.2. Planning Domain Definition Language (PDDL)

PDDL [24] is a language for describing planning domain and problems that is widely supported by planning tools, providing several features, depending on the degree of detail of the domain being specified. PDDL was proposed and maintained in the context of the International Planning Competition [40], with each new edition of IPC adding new features to the language specification. In PDDL, the planning domain and its associated problems are described in separated text files. The domain file contains the domain requirements (i.e., the features the planner must implement in order to work with that domain), the domain objects, the predicates and the actions. Depending on the requirements, the description may contain other elements, e.g., axiom descriptions. Examples of requirements frequently found in domain descriptions are:

`:strips`	Indicates a STRIPS-like domain;
`:typing`	Support to object types;
`:equality`	Support "=" as built-in predicate;
`:disjunctive-preconditions`	Allow `or` in goal descriptions;
`:existential-preconditions`	Allow `exists` in goal descriptions;
`:universal-preconditions`	Allow `forall` in goal descriptions;
`:conditional-effects`	Allow `when` in action effects;
`:adl`	`:strips + :typing + :equality + :disjunctive-preconditions + :quantified-preconditions + :conditional-effects.`

In Figure 2-4 we show an example of description in PDDL for the Logistics domain, adapted from the test cases in the PDDL4J library [69]. This domain, as the name suggests, describes a scenario where packages need to be delivered across cities and airports by using either trucks or airplanes. Besides STRIPS support, this domain requires the use of typed objects, as can be seen in the `(:types...)` statement. Following the domain types are the predicates – `(:predicates...)` –, which in this example are used to determine whether a package, truck or airplane is in some given location. The domain actions are described in the sequence, with their parameters, preconditions and effects. Note the preconditions and effects are logic formulations over the action parameters using the predicates described earlier in the file. Action `load-truck`, for example, takes a package, a truck and a location as parameters, and has as precondition that both the truck and package are in the location specified. As its effect, the action `load-truck` takes the object from the location and puts it in the truck.

An example of planning problem using the Logistics domain is showed in Figure 2-5. The problem description brings the objects that exist in this problem instance in the `(:objects...)` statement, declared with the types from the domain file (the `mxf` is an object of type `package`, and `airplane1` is an `airplane`). In the sequence, comes the initial state description – `(:init...)` – which in this case puts airplanes and packages in their initial positions (the `mxf` package, for example, is at `par-airport` airport). The goal state defined in the `(:goal...)` part states the final positions of the packages, which for the `mxf` package is the airport `bos-airport`.

```
(define (domain logistics)
  (:requirements :strips :typing)
  (:types
      package location vehicle - object
      truck airplane - vehicle
      city airport - location)

  (:predicates
      (at ?vehicle-or-package - (either vehicle package)  ?location - location)
      (in ?package - package ?vehicle - vehicle)
      (in-city ?loc-or-truck - (either location truck) ?citys - city))

  (:action load-truck
      :parameters
             (?obj - package  ?truck - truck  ?loc - location)
      :precondition
             (and   (at ?truck ?loc) (at ?obj ?loc))
      :effect
             (and   (not (at ?obj ?loc)) (in ?obj ?truck)))

  (:action load-airplane
      :parameters
             (?obj - package ?airplane - airplane ?loc - airport)
      :precondition
             (and (at ?obj ?loc) (at ?airplane ?loc))
      :effect
             (and   (not (at ?obj ?loc)) (in ?obj ?airplane)))

  (:action unload-truck
      :parameters
             (?obj - package ?truck - truck ?loc - location)
      :precondition
             (and  (at ?truck ?loc) (in ?obj ?truck))
      :effect
             (and   (not (in ?obj ?truck)) (at ?obj ?loc)))

  (:action unload-airplane
      :parameters
             (?obj - package ?airplane - airplane ?loc - airport)
      :precondition
             (and   (in ?obj ?airplane) (at ?airplane ?loc))
      :effect
             (and (not (in ?obj ?airplane)) (at ?obj ?loc)))

  (:action drive-truck
      :parameters
             (?truck - truck ?loc-from - location ?loc-to - location ?city - city)
      :precondition
             (and   (at ?truck ?loc-from) (in-city ?loc-from ?city)
                    (in-city ?loc-to ?city))
      :effect
             (and   (not (at ?truck ?loc-from)) (at ?truck ?loc-to)))

  (:action fly-airplane
      :parameters
             (?airplane - airplane ?loc-from - airport ?loc-to - airport)
      :precondition
             (at ?airplane ?loc-from)
      :effect
             (and   (not (at ?airplane ?loc-from)) (at ?airplane ?loc-to)))
)
```

Figure 2-4. Example of PDDL description for the Logistics domain.

```
(define (problem pb1)
  (:domain logistics)
  (:objects mxf - package
          avrim - package
          alex - package
          jason - package
          pencil - package
          paper - package
          april - package
          michelle - package
          betty - package
          lisa - package
          airplane1 - airplane
          airplane2 - airplane
          lon-airport - airport
          par-airport -  airport
          jfk-airport -  airport
          bos-airport -  airport)
  (:init (at airplane1 jfk-airport)
       (at airplane2 bos-airport)
       (at mxf par-airport)
       (at avrim par-airport)
       (at alex par-airport)
       (at jason jfk-airport)
       (at pencil lon-airport)
       (at paper lon-airport)
       (at michelle lon-airport)
       (at april lon-airport)
       (at betty lon-airport)
       (at lisa lon-airport)
       )
  (:goal (and
        (at mxf bos-airport)
        (at avrim jfk-airport)
        (at pencil bos-airport)
        (at alex jfk-airport)
        (at april bos-airport)
        (at lisa par-airport)
        (at michelle jfk-airport)
        (at jason bos-airport)
        (at paper par-airport)
        (at betty jfk-airport)
        )
       )
  )
```

Figure 2-5. Example of problem description in PDDL for the Logistics domain.

2.5. SUMMARY AND DISCUSSION

In this Chapter, we introduced the fundamental concepts used in the development of this Thesis: Service Oriented Computing – including Service Composition – and Automated Planning. Semantic Web concepts were presented in order to educate the reader on the subjects of ontologies and service descriptions, although descriptions of services using standard ontology languages (such as OWL-S) were not used in this Thesis. The next Chapter brings the State of the Art on Automated Service Composition techniques.

Chapter 3 Automated Composition Techniques

In this Chapter, we present a compilation of the most relevant approaches in literature for Automated Service Composition. These works where organized according to the techniques they use to address the composition problem. In the end of the chapter, we present a discussion on the works presented here.

3.1. Planning Graph

The work in [97] proposes a service composition technique based on matching the services input and output parameters described in their interfaces. In that approach, services are described using the early service ontology language DAML-S [4] (current OWL-S [89]). The composition developer needs to provide the set of input and output parameters for the composition and the algorithm try to find a sequence of service invocations that, starting with the input parameters, provides the intended outcomes. The algorithm performs a forward, looking for services that have the input parameters matching all or some of the inputs initially provided. Then the outputs of the services found are added to the set of available inputs and a new discovery step is performed. The process is repeated until all original outputs are returned by discovered services or in case no progress is achieved (no more services are found that match the available inputs). The result of these steps is akin to a weighted Planning Graph , where the services found are connected according to the dependence of input/output parameters. During the process of service discovery, a similarity value is used to measure how compatible the available parameters and those required by the services are (i.e., if the data types are identical there is an exact match with value 0, higher values otherwise). The similarity is used to weight the links between services. The next step consists of identifying the final composition in the Planning Graph. The graph contains subparts that lead to no valid solution (i.e., none of the outputs), but also several solutions may exist. The proposal in [97] uses the Bellman-Ford algorithm to find a sequence of services in the graph for the simpler one-input/one-output case. The algorithm tries to find the path with minimal total weight from the input to the output parameters. For the more general case where several input and output parameters are specified, the authors of [97] suggest applying the Bellman-Ford algorithm to inputs and outputs pairwise, and then combining the paths found; no further details on this procedure were presented. Values other than the similarity measure can be used as weight, for example, QoS measures (response time, availability, etc.), as proposed in [97]. An approach very similar to [97] is presented in Arpinar et al. in [5] while adopting an incremental, semi-automated composition strategy, where the user is asked at each discovery iteration which available input parameters should be used or discarded. The authors of [97] also describe a distributed version of the composition algorithm using peer-to-peer (P2P) technologies.

In the work in [95], Yan and Zheng argue that traditional approaches for automated service composition do not scale well for problems of significant size. Based on this premise, they employ a Planning Graph-based approach to take advantage of the polynomial complexity of graph construction. The services in this work have OWL descriptions attached to them, and the semantic information on the service's inputs and outputs is used for matching the services during the expansion of graph. Instead of performing a backward search in the Planning Graph once a level containing the goal outputs is found, the authors of [82] focus on minimizing the number of unnecessary services added to the graph during its expansion. For that end, they apply several strategies, for example, not adding a service at a given level of the graph if its outputs were already provided by another service in the same level or any level above. In the end, their algorithm renders a composition that fulfills the goals but can still contain redundant services. Nonetheless, the use of only the services' input and outputs, as described in [82], implies that no mutual exclusion relations between services exist in the resulting Planning Graph. In such conditions, the standard backward search can be performed quickly since it does not backtrack, diminishing the need for strategies presented in related research used to limit the graph size.

3.2. LOGIC PROGRAMMING

In [57], McIlraith et al. proposed the use of Situation Calculus [72] and the logical programming environment Golog for implementing service composition. Situation Calculus is a logical formalism suitable for modeling composition problems since it supports the idea of actions – with preconditions and effects – that once executed trigger the change from one situation to another (situations are akin to states but formally represent sequences of action invocations). In [57], the composition processes is based on a set of manually written generic procedures and a set of user constraints on desirable states. The procedures contain actions equipped with preconditions, i.e., state the system should be before action execution. An order operator ":" is introduced to state that one action should precede another (e.g., "$a: b$"), allowing other actions to be inserted in between automatically as long as the actions' preconditions hold (for example, "$a: b$" becomes "$a; x; y; z; b$"). The operator is used by the modified Golog interpreter to interpolate actions in the generic procedures in order to meet the specification of the composition developer. An automated planning strategy using the A* algorithm is used to find the intermediate actions. The composition produced by the algorithm may contain sensing actions, which sense the current system state, for example, in case the value of some variable is unknown. The sensing actions trigger the invocation of services in composition-time, (the authors of [57] assume most Web services are information-providing services). Sohrabi et al. [78] extended the approach in [57] with developer preferences on the intermediate states the service composition could assume. For example, in a Travel scenario, the developer can specify that the composition should not book the air ticket before having a confirmation that the hotel room was reserved. These approaches, however, required the developer to look at the pool of generic procedures available and find the one the fits his/her needs, what hinder their usability.

3.3. Hierarchical Task Networks

At the core of work in [92], Wu et al. used the SHOP2 planner [62] to automatically compose service based on services' inputs, preconditions and effects (modeled as del-conditions and add-conditions, which tell the literals that will become false and true after execution, respectively). The SHOP2 planner is a Hierarchical Task Network planning tool that takes advantage of the inherent hierarchical structure of plans: high-level tasks can be decomposed into a few smaller abstract tasks, which are further divided successively until atomic tasks are found [62]. The authors used the Process Model in DAML-S descriptions (currently OWL-S Process Model) of the services to derive abstract methods used by HTN planner. The Process Model of a service tells how the clients and the service have to interact, i.e., which steps the service follows and what it expects from the clients at each step. Conditional branching and iteration, among other control constructs, are supported in the Process Model. The authors of [92] separate tasks into informative (e.g., "consult ticket price") and world-altering services (e.g., "book ticket"). During planning, informative services are invoked to allow the planner to make decisions on which execution path to follow. The final plan, however, contains only world-altering services, which are not executed in planning time.

Lin et al. present in [53] an approach similar to that in [92] – combining HTN with OWL-S Process Models –, but adds support to user preferences. The preferences are specified using PDDL3 [23], and allow the developer to prioritize certain system states, e.g., telling which subgoals have to be achieved first. Their algorithm, called SCUP (Service Composition with User Preferences), combines HTN with best-first search, and tries to minimize the penalty of violating the preferences defined by the developer. Klusch et al. [46] also present a HTN based approach where the WSC problem is translated to a PDDL description and an HTN planner is used to solve the problem. All these approaches, however, have the drawback of using the Process Model of OWL-S descriptions, which is more complex and likely to be less frequently available than the Service Profile, which describes the service observable behavior (i.e., inputs, outputs, preconditions and effects). Also, while the use of the Process Model makes sense for services with complex interactions – i.e., services that require the client to make several invocations to obtain the end result –, it is less useful for more common request-response services.

3.4. CONTINGENCY PLANNING

Agarwal et al., propose Synthy [1], a two-stage composition approach. Synthy groups services in types, in order to increase scalability. The first stage consists of finding a composition based on service types, i.e., instead of connecting the services directly, abstract types are used as placeholders. The authors use the Planner4J tool, for performing the composition itself. Services are equipped with preconditions and effects described in OWL-S. Preconditions and effects are defined as sets of literals that hold before and after service execution, respectively. The service's preconditions and effects are used to group services into types: a service belongs to a type if its preconditions are more generic than the type's preconditions and its effects are more specific than the type's effects. The Planner4J is planning framework supporting several planning strategies, including classical algorithms and HTN. It can also compute several solutions to a single request, allowing for alternative compositions in case of failure. In [1] a contingency planning strategy was used. Contingency planning assumes that the system state may be unknown at certain points, in opposition to the complete knowledge premise of classical planning. To handle the uncertainty, the planning algorithm adds sensing actions and decision points in the plans it generates. The sensing actions collect information from the environment in order to make clear the current state of the system (for example, the value of some variable that is unknown), what enables a decision of which plan path to follow next during plan execution. The second stage of the composition process consists of selecting at runtime the instances that are going to be executed based on the non-functional goals, for example, response-time. This approach decouples functional and non-functional requirements of the composition, also providing greater flexibility when executing the plans. According to the authors, the overall complexity of the approach is $O(K\alpha^{\lambda})$, where K is the number of alternative plans to be computed, α is number of service types and λ the maximum number of actions per plan.

A similar approach is presented by Akkiraju et al. [2]. The authors called it SEMAPLAN, and it combines AI Planning with Information Retrieval techniques for improving planning speed. A forward planning heuristic based on the A* algorithm is used. Services are modeled with pre- and postconditions (effects) in the same manner as in [1] (sets of literals), and similarly several plans are generated. Indexing techniques are applied to services in order to speedup service discovery by the planner. Authors also gave extra attention to multi-domain ontological compatibility, which differentiates them from the work in [1].

3.5. Planning as Model Checking

Traverso et al. [83] use the Process Model of OWL-S descriptions associated to services, as in [92] and [53], in order to perform service composition. The Process Model of each service is mapped into a nondeterministic (i.e., an action may have several outcomes), partially observable (i.e., only part of the system state is visible externally) labeled transition system. A Model Checking planning strategy is used to interleave the services transition systems and achieve the goals specified by the user. The work in [83] also supports complex goal descriptions, using a language created for that purpose called Eagle. Complex goals allow the user to define, for example, the primary goal of the composition and a secondary goal in case the primary goal fails. Executable BPEL code is generated as the final result of the composition process. This approaches suffers with the same drawbacks of [92] and [53] of depending on the Process Model of services for its composition procedure.

3.6. Service Dependency/Causal Matrix

The idea of preprocessing the service repository is explored by Lécué et al. in [50]. A so-called Causal Link Matrix is built before any request for service composition. The CLM opposes input (rows) and output (columns) parameters and has all available parameters at each dimension. Each matrix cell relating two parameters contain a list of services that connect them (i.e., one as input and the other as output of the service), each entry followed by the semantic similarity of the actual service output to the cell output. Once built, the CLM is used in the composition process, which traverses the matrix from inputs to outputs to build the service composition. The complexity of building the CLM is $O(m^2n)$, where m is the total number of parameters and n the number of services in the system, plus the cost of updating it every time a service is added or removed. Omer et al. [66] also considered the use of a dependency matrix (DM) for computing the composition. In [66], the DM is computed on demand based on the services that match the input and outputs provided. The work in [66] takes for granted the task of finding theses services, not taking the discovery complexity into account.

3.7. AUTOMATED THEOREM PROVING

More recently, McCandless et al. [55] proposed the use of automation to generate semantic service descriptions prior composition, in contrast to previous proposals that assume a manual description process. In [55], existing tools were used for extracting ontologies from WSDL descriptions and aligning them in order to have common dictionaries among services. The OWL-S ontology was used to describe the service's semantics. For the service composition itself, the service creation query (input and output parameters) is translated and fed into the automated proof tool Prover9 [71], which use the service descriptions as axioms to find a proof that connects the inputs and to the outputs provided. The proof is later translated into a BPEL script for deployment and execution.

3.8. GENETIC ALGORITHMS

A different approach for service composition is proposed Rodríguez-Mier et al. [75], where the authors try to address the lack of control structures in the compositions found by other approaches. The control structures supported by the algorithm in [75] are: choice (nondeterministic), loops, sequences, splits (parallelism) and splitJoin's (parallelism with synchronization at the end). The authors of [75] estimated the complexity of finding such complex composition as being $O(n!\,m^{2d})$, where n is the number of services, m is the number of control structures and d is the maximum depth of the composition (longest execution path). In order to tackle this complexity, they used an evolutionary approach based on genetic algorithms. A context-free grammar containing the control structures was used to model valid composition skeletons. These skeletons were generated randomly (with limited depth) and formed the initial population of the genetic algorithm. Services from the repository were attached to the skeletons so that individual control structures had their input and output constraints filled in and complete the compositions. The fitness of each individual of the population (i.e., composition skeleton with attached services) depended on the functional adequacy – whether the services and control structures combined provided the desired outcomes –, and also on the composition depth and number of services. Several generations were necessary to obtain the final composition, with the best individuals being selected pairwise to generate offspring (mixed compositions) subject to random mutations (e.g., control structures replaced). At every round, the worst individuals of the population were eliminated to keep the number of individuals constant. Experiments with several scenarios – with queries requiring from simple sequences to complex sequence-split-splitJoin combinations – were presented (the repositories OWL-S TC [67] and from the Web Services Challenge (WSC) 2008 [90] were used). In the experiments, with populations limited to 100 individuals and 100 generations, the most complex cases took around 1min30s to complete in a quad core processor, 8GB machine. The compositions found were correct in all cases. Although yielding good results, the scenarios tested required only sequential and parallel execution that could be obtained by direct algorithms (for example, combining execution path and letting non-dependent services be executed in parallel). No concrete scenarios using choice (if-like constructions) or loops were presented.

3.9. Summary and Discussion

As seen in the previous sections, the most common approach to automated composition employs AI-based planning tools (for example, [97], [1], [2], and [92]), in order to find execution plans that, in the end, will make up the service composition. Most of the approaches in this chapter work similarly, requiring the specification of initial and final states in order to derive a path of services that meet these requirements. The services should also provide some kind of semantic description its effect in the system state in order for them to work.

Several works listed in the previous section compose services using only the services' input and output parameters described with semantically enhanced types (see [97], [82], [50], [66] and [55]). The new composition is described in terms of what it should receive as input and what result it should render at the end. Services are connected together so that one service output(s) can be used as input(s) to the next service (or any following service) and so on, provided that parameter types match. In this kind of approaches, the more meaning the parameters types carry, the more precise will be the composition found. For example, when the composition procedure is looking for a service that accepts an $AccountId$ and returns the $AccountBalance$, the correct (hypothetical) $GetAccountBalance$ service is likely to be found. On the other hand, a composition process based only on primitive types can combine completely unrelated services. For the previous example, assuming $AccountNumber$ is a character string and $AccountBalance$ is a double-precision Real, services such as $GetLocationTemperature$ or even $ParseStringToDouble$ are possible matching candidates. A shared library of semantically described data types is an important puzzle piece in these approaches, and ontologies are the natural tool for the job. While the Web Ontology Language (OWL) provides the means for describing concepts and relating them together, its service extension OWL-S allows for binding service parameters to those concepts. Further, a semantic-based discovery service is needed in order to find services that individually meet the requested input and output types. The discovery procedure compares the data types from the query (those being looked for) and the available services looking for semantic similarities: exact matches (same concept in the ontology), sub- or superclass matches, sibling matches (specialization of same superclass), etc. Partial matches (non-exact) are useful to generate approximate compositions. Proposals for semantic service discovery are presented in [31] and [45], among others.

Service composition using preconditions and effects – besides input and output parameters –, is a very powerful approach that can, in theory, find correct-by-construction execution plans that meet the developer's requirements This approach, employed by [57], [78], [92], [53], [46], [1], [2], [83] and [75], has a few practical obstacles in order to be put in production scenarios. There is the need for complete formal descriptions of each service, sometimes requiring a detailed description of service interactions as in [57] and [83], for example. These descriptions must be as detailed and as correct as possible in order to ensure that sound compositions are rendered. Finally, the ability to write descriptions using logic formalisms is not a typical skill of service developers, more accustomed to object-oriented languages and graphical IDEs.

Composition methods based on simple I/O (Input/Output) matching share some drawbacks with the most complete IOPE (Input, Output, Precondition and Effects) methods. I/O composition is not as powerful as IOPE approaches since only the type of parameter is matched, not the parameter state. I/O matching, on the other hand, can be mimicked by state-based approaches with relatively simple logic tweaks, for example, the service's precondition may require the availability of input parameters (e.g., $available(AccountNumber) = true$), while effects state that the service's outputs are available after execution. In practical situations, however, I/O-based composition seems more appealing to developers or even non-technical users given its simplicity and similarity to a production chain. For this reason we opted to focus on automated composition techniques based on input and output parameters; the need for more sophisticated, pre- and postcondition-based composition is a subject for future work.

Chapter 4 Reverse Engineering Compositions

Semantic description of services is a key part of the automated composition process. The semantics allow the algorithms to decide whether a service is appropriate for a specific task or not. As discussed in Chapter 1, however, semantic descriptions are not commonly found with services, which provide only syntactic descriptions of their interfaces. In order to tackle this problem we present in this chapter a process for gathering semantic information from existing services based on their usage patterns. Our process consists in analyzing a repository of existing service compositions – the myExperiment project [26] –, from which the relationships between services can be extracted to be later used for creating new compositions. The semantics we are interested here are new knowledge about the services and their relationships that can help reducing the search space and produce new compositions more similar to the ones created by human developers.

This Chapter begins introducing the formal definitions of service and composition used in the remaining of this Thesis. After that, the information extracted from the composition repository is described in details. Then the reverse engineering process is applied to an existing repository of scientific compositions (or "workflows"). This repository is the basis for the evaluations and experiments presented in this Thesis. A summary of the chapter and discussion follows after that.

4.1. Composition and Service Models

For the purpose of this Thesis, a service is defined as follows:

Definition 4-1 (Service) A *service* is a tuple $S = \langle Id_S, I_S, O_S, typeOf_S \rangle$ where:

- Id_S is an unique service identifier;
- $I_S = \{p_0, p_1, \dots, p_n\}$ is the set of input parameters;
- $O_S = \{q_0, q_1, \dots, q_m\}$ is the set of output parameters;
- $typeOf_S: (I_S \cup O_S) \rightarrow T$ is a function that maps a parameters into a parameter type from the set of types T.

□

The service identification Id_S allows different services with the same number and types of parameters to be used. The set of all services is identified by ϕ. A service can be invoked more than once in the same composition and, therefore, we need to model individual service invocations. A service invocation is an instantiation of a service and its parameters and represents the act of calling the service.

Definition 4-2 (Service Invocation) An *service invocation* is defined as $inv = \langle Id_{inv}, S_{inv}, P_{inv}, Q_{inv} \rangle$, where:

- Id_{inv} is an unique invocation identifier;
- S_{inv} is the service being invoked;
- $P_{inv} = instance_{inv}(I_{S_{inv}})$ is the set of input parameter instances;
- $Q_{inv} = instance_{inv}\ (O_{S_{inv}})$ is the set of output parameter instances.

□

The construction $instance_{inv}(R) = \{(id, p) : p \in R\}$ provides instances of parameters in the set R associated with some invocation inv or to a composition C. Parameter instances are occurrences of a service parameter associated to a given invocation of that service. In order to identify parameter instances uniquely we use an ordered pair containing the invocation (or composition) identifier, id, and the parameter itself.

The data flows between service invocations through parameter assignments. An assignment connects an output parameter instance of a service invocation to an input parameter instance of another invocation, meaning that the value returned by the first invocation will be passed as an argument to the second. Invocations can be executed in parallel as long as they do not depend on each other, that is, there is no assignment from one invocation's output to the input of the others.

Definition 4-3 (Parameter Assignment) A *parameter assignment* is an ordered pair $assign = (source, sink)$, where $source = (id_{source}, param_{source})$ and $sink = (id_{sink}, param_{sink})$ are parameter instances associated to two service invocations or the composition itself, and $id_{source} \neq id_{sink}$.

□

Finally, a service composition C (or simply "composition") is defined akin to a service invocation (Definition 4-2):

Definition 4-4 (Service Composition) A *service composition* C is defined by the tuple $C = \langle Id_C, S_C, P_C, Q_C, invocations_C, assigns_C \rangle$, where:

- Id_C is the composition unique identifier;
- $S_C = \langle Id_{S_C}, I_{S_C}, O_{S_C}, typeOf_{S_C} \rangle$ is the service associated with composition C;
- $P_C = instance_C(I_{S_C})$ is the input instances of C;
- $Q_C = instance_C(O_{S_C})$ is the output instances of C;
- $invocations_C$ is the set of invocations that comprise C;
- $assigns_C$ is the set of parameter assignments that connect the invocations in C.

□

Since a composition is also a service, it has a service S_C associated with it that defines the composition interface, i.e., its input and output parameters. As the invocation, a composition also contains the associated parameter instances, P_C and Q_C, which serve as the data entry and exit points. The composition extends the invocation model by adding information about its internals: the set of invocations it contains, $invocations_C$; and the assignments connecting them, $assigns_C$. Assignments also connect the input instances of the composition to the input of an invocation, or the output instance of one invocation to an output of the composition.

Definition 4-5 (Valid Composition) A composition C is said to be a *valid composition* if and only if all the following hold:

1. All parameter assignments involve only invocations within the composition, or the composition itself:
$$\forall (source, sink) \in assigns_C, \qquad valid_C(id_{source}) \wedge valid_C(id_{sink})$$
Where:
   - $source = (id_{source}, p)$ and $sink = (id_{sink}, q)$;
   - $valid_C(id) = [(id = id_C) \vee (\exists inv \in Invs_C : id = id_{inv})]$.
2. All output parameters of C are assigned:
$$\forall q \in Q_C, \exists (source, sink) \in assigns_C, sink = q$$
3. All service input parameters are assigned for every invocation:
$$\forall inv \in invocations_C, \forall p \in P_{inv}, \exists (source, sink) \in assigns_C, sink = p$$
4. There are no multiple assignments for one given input parameter instance in the same invocation:
$$\forall a, b \in Assigns_C, a = (source_a, sink_a), b = (source_b, sink_b),$$
$$sink_a = sink_b \rightarrow source_a = source_b$$

5. There are no self-assignments, i.e., no input and output instances of the same service invocation are assigned to each other:
$$\forall(source, sink) \in assigns_C, id_{source} \neq id_{sink}$$
6. All assignments respect type compatibility (type compatibility is discussed in more detail in the next section):
$$\forall(source, sink) \in assigns_C, source = (id_{source}, p),$$
$$sink = (id_{sink}, q), compatible(p, q);$$
7. The composition C is a Direct Acyclic Graph (DAG), containing no cycles involving invocations, i.e., an invocation cannot precede itself. A formal definition of precedence is given in Section 4.2.2.

□

The concept of valid composition is useful for filtering out defective compositions present in the composition repository during the reverse engineering process, which is described in the next section.

4.2. Extracting Information from Compositions

4.2.1. Parameter Analysis

The goal of parameter analysis is to determine when a parameter – usually the output of a service – can be passed as input to another service. Parameter compatibility is based on the types involved, and in strongly typed languages such as Java and C/C++, it is solved easily in compilation time based on the existing type hierarchy. However, when dealing with services from different sources, it may be difficult to establish such compatibility. In our case, we have a set of existing compositions and we want to build an approximate type hierarchy from it. The type hierarchy, in its turn, can be used by developers as an automated composition process to derive new compositions.

In typical Object Oriented languages, a parameter of a more specific type t_{sub} can be passed as input parameter of a more generic type t_{super}, i.e., provided that t_{sub} is a subtype of t_{super} (directly or inderectly), t_{sub} is compatible with t_{super}. This relationship can be seen as a set containment relationships where $t_{sub} \subseteq t_{super}$, that is, all parameters that belong to set t_{sub} also belong to set t_{super}.

In order to extract the relationships between parameter types from the compositions we analyzed the assignments of service outputs into service inputs and added the corresponding compatibility relationships to the type hierarchy. Figure 4-1 illustrates this process for two sample compositions. The resulting type hierarchy, inferred from the assignments between services, is shown in the right-hand side of Figure 4-1. In this figure, Service 1 has one output parameter of type "E" which is passed to services 2 and 3 as parameters of types "D" and "B" respectively; the other service connections follow suit.

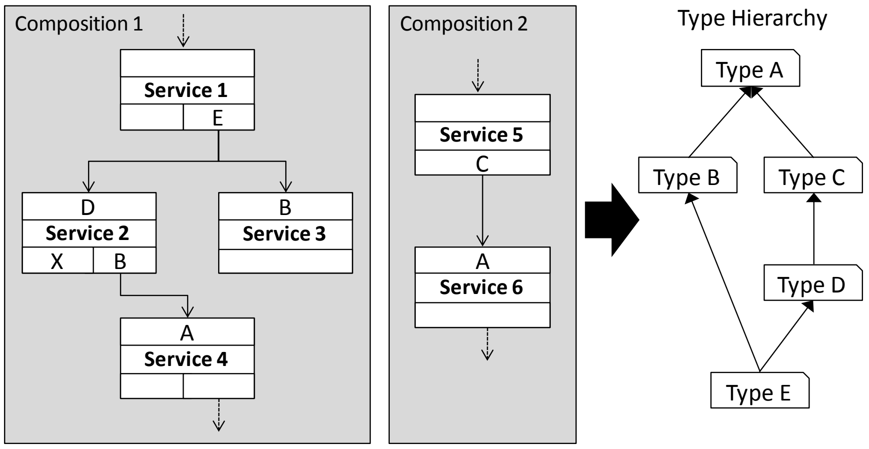


Figure 4-1. Type hierarchy generated from 2 sample service compositions.

Formally, the type hierarchy can be defined by the predicate $subtypeOf(T_1, T_2)$, where T_1 and T_2 are types, which is read "T_1 is a subtype of T_2". The predicate $subtypeOf()$ is defined using function $number\_assign: \Gamma \times \Gamma \rightarrow \mathbb{N}$, which provides de number of occurrences of assignment from type T_1 to type T_2 in all compositions in the database. This function is defined as:

$$subtypeOf(T_1, T_2) = \begin{cases} true, & assignCount(T_1, T_2) > K \\ false, & otherwise \end{cases} \tag{4-1}$$

Where:

$$assignCount(T_1, T_2) = \sum_{C \in \omega} assignCount_C(T_1, T_2) \tag{4-2}$$

$$\begin{aligned} & assignCount_C(T_1, T_2) \\ & \qquad = |\{(source, sink) \in assigns_C : typeOf(param_{source}) \\ & \qquad = T_1 \wedge typeOf(param_{sink}) = T_2\}| \end{aligned} \tag{4-3}$$

Equation (4-3) gives the number of assignments between the given types for a single composition C. The condition for types T_1 and T_2 to have a sub/super-type relantionship is that the number of occurrences of assignments from T_1 to T_2 being greater than K, an integer constant defined by the developer. In our experiments K equals zero, meaning that a single assignment in only one composition implies that the types are related. A more conservative developer could set K to 1 or 2, for example, so that wrong assignments (assumed to be less frequent that correct ones) would not pollute the type hierarchy, but with the expense of some correct assignments being left out. Another possibility – yet to prevent mistaken type relations – would be considering only assignments that occur in more than one composition (or, if the information is available, in compositions from different users). These variations, however, were not investigated in this Thesis.

4.2.2. Service Analysis

The purpose of the service analysis is to extract meaningful relationships between services based on their usage in the composition database. We are especially interested in relationships that can be used and render improvements in the automated creation of new composition.

The first relationship explored relates services used in the same compositions. Two services are said to be "related" if they are present in one more compositions. Let Ω be the set containing all compositions in our repository, the predicate $related(S_1, S_2)$ is defined as follows:

$$related(S_1, S_2) = \begin{cases} true, & \exists C \in \Omega : S_1, S_2 \in services_C \\ false, & otherwise \end{cases} \qquad \textbf{(4-4)}$$

Where:

$$services_C = \{ S_{inv} : inv \in invocations_C \} \qquad \textbf{(4-5)}$$

From this definition follows that $related(S_1, S_2) \leftrightarrow related(S_2, S_1)$. This relation defines an undirected graph with the services as vertices. Each new composition that is analyzed creates a clique in this graph, i.e., all services in the composition are connected to each other.

Another interesting relationship is the one where services have direct dependencies within compositions, i.e., services whose input and output parameters are directly connected. This relationship is given by the predicate $dependsOn(S_1, S_2)$, read "S_1 depends on S_2", which applies to the whole set of compositions. The relation holds even if the dependency happens in only one composition. $dependsOn(S_1, S_2)$ is formally described as:

$$dependsOn(S_1, S_2) = \begin{cases} true, & S_2 \in depend_{services}(S_1) \\ false, & otherwise \end{cases} \qquad \textbf{(4-6)}$$

Where:

$$depends(S) = \bigcup_{C \in \Omega} depends_C(S) \tag{4-7}$$

$$depends_C(S) = \{S_{inv} : inv \in depends_C^{invs}(S)\} \tag{4-8}$$

$$depends_C^{invs}(S) = \bigcup_{i \in invs_C^S} dependenciesOf_C(i) \tag{4-9}$$

The function $depend_{services}(S)$ in equation (4-6) provides the set of services that S depends on for all compositions; $depend_{services,C}(S)$ does the same for a specific composition C while $depend_{invocations,C}(S)$ gives the invocations instead of the services. $invs_C^S$ is the set of invocations of service S in composition C. $dependenciesOf_C(i)$ lists the invocations that invocation i depends on in the composition.

A stricter version of this relationship, $alwaysDependsOn(S_1, S_2)$, includes only the cases where the dependency manisfets itself in all compositions. In this case, we have alternative versions of $depend(S)$ and $depend_C^{invs}(S)$ where the unions are replaced by intersections.

Another relationship explored is the precedence relation, $precedes(S_1, S_2)$ ("S_1 precedes S_2"), that happens when service S_1 appears in the execution sequence of the composition before service S_2. Since the composition, as defined in Section 4.1, does not carry service ordering explicitly, precedence is defined in terms of the dependency of input parameters. In other words, S_1 precedes S_2 if, for any given composition, any input of S_2 depends on an outputs of S_1 (i.e., $dependsOn(S_2, S_1)$ holds) or S_1 precedes any of the services S_2 depends on. It can be described as:

$$precedes(S_1, S_2) = \begin{cases} true, & \exists C \in \Omega : predeces_C(S_1, S_2) \\ false, & otherwise \end{cases} \tag{4-10}$$

Where:

$$precedes_C(S_1, S_2) = \begin{cases} true, & dependsOn(S_2, S_1) \vee \left(\exists S_3 \in depend_C(S_2) : precedes_C(S_1, S_3)\right) \\ false, & otherwise \end{cases} \tag{4-11}$$

As for the $dependsOn()$ relationship, the precedence relation can be made more restrictive if we consider that the relation holds only if the precedence happens on all compositions, not being sufficient holding for only one composition as before. This new relation, $alwaysPrecedes(S_1, S_2)$, is defined as:

$$alwaysPrecedes(S_1, S_2) = \begin{cases} true, & \forall C \in \omega : predeces_C(S_1, S_2) \\ false, & otherwise \end{cases} \tag{4-12}$$

The relation in equation (4-12) can be very helpful in automated composition since it establishes a precondition-like relationship between the services, which can be used to prune the search space and produce more accurate solutions. A relaxed version of this relation can be defined with the help of the $precedence(S_1, S_2)$ function, $precedence: \phi \times \phi \rightarrow \mathbb{R}^+$, defined in equation (4-13), which gives the percentage of compositions where S_1 precedes S_2 (compared to the number of compositions that contain S_2). Then, a service S_1 can be said to be a prerequisite for S_2 if $precedence(S_1, S_2)$ passes a given threshold.

$$precedence(S_1, S_2) = \frac{|\{C \in \Omega : precedes_C(S_1, S_2)\}|}{|\{C \in \Omega : S_2 \in services_C\}|} \tag{4-13}$$

4.3. Case Study: the myExperiment Repository

In this section, we describe the application of the reverse engineering process described in Section 4.2 in a real service composition repository: the myExperiment social network for scientific workflows. In the myExperiment repository, service compositions are called workflows, and therefore the terms "composition" and "workflow" are used interchangeably throughout the remaining of this Thesis. We used the workflows from the myExperiment repository to validate our algorithms and evaluate automated planners from the literature.

4.3.1. The myExperiment Workflow Repository

The myExperiment project [26], [60] provides a social network of scientists that collaborate and publish intensive data-processing workflows, especially in the biological sciences field. The project aims at facilitating the creation and exchange of scientific workflows among scientists for leveraging their research. More than 5000 members have joined the project and published over 2000 workflows since 2007 [60]. myExperiment is part of a group of partner projects that emerged from the myGrid initiative [61], which also includes BioCatalogue [10] and Taverna [81], among others. These projects have complementary roles in the scientific workflow production chain. The BioCatalogue project offers a repository of services for biological sciences. These services can be tied together to form new workflows using the Taverna workbench. The new workflows are published in the myExperiment website, which connects researchers and promotes knowledge exchange. The next section details the Taverna tool and its workflow format.

4.3.2. The Taverna Workbench and Workflow Format

The Taverna Workbench [39], [81] is a graphical IDE for developing scientific workflows. It integrates with BioCatalogue and myExperiment, allowing service information retrieval and publishing the workflows from within the development environment. The IDE uses its own XML-based formats: Taverna 1 (T1, older) and Taverna 2 (T2, new format). This Thesis only uses and refers to workflows in the T2 format. Figure 4-2 shows a screenshot of the Taverna Workbench with a workflow being edited.

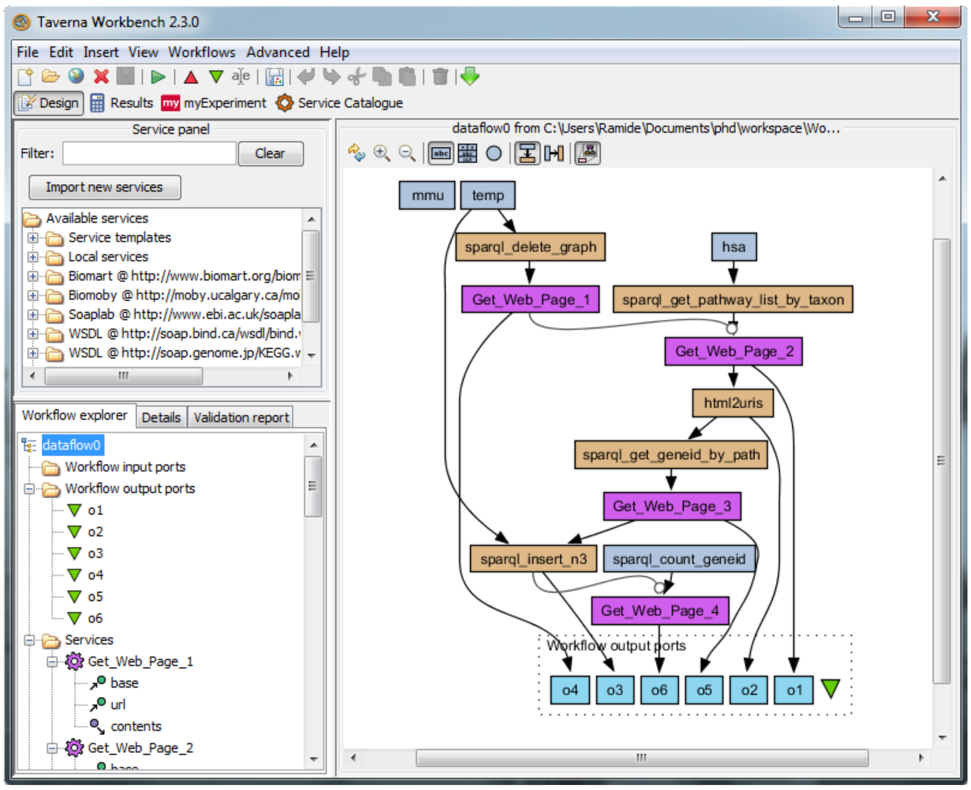


Figure 4-2. The Taverna Workbench.

Both Taverna formats are dataflow oriented, i.e., they focus on the data connections between processing elements instead of the precise order of execution of the tasks, as in most composition languages. Dataflow compositions (e.g., a T2 workflow), however, can be easily translated into imperative representations (e.g., a BPEL script) as long as the input and output dependencies between tasks are respected in the final task execution sequence.

A Taverna Workflow is comprised of one or more Dataflows; one main dataflow and other auxiliary dataflows that can be called from within the main one. Dataflows, in their turn, contain Processors and Datalinks. Processors are elements responsible for executing data processing tasks, while a datalink represents the connection between processors through their ports. The datalink Source is an output port of a processor and the datalink Sink is an input port of another processor. Input and output ports are the data entrance and exit points to and from the processor. Dataflows also contain ports, which are connected to processor ports via datalinks. Inside a Processor, there exist one or several Activities. Whenever two or more activities are present in a processor, this is for the purpose of failure handling or parallelism. These cases, however, were not taken into account in the analysis of the workflows.

Activities are the *de facto* atomic tasks of the Taverna workflow and encapsulate different task types, one of them being Web Service invocation. Other types of activities include script activities (java-based), querying biology databases, XML processing, among others. Other activity types can be created by third-party developers. All activities have to map the specific task inputs and outputs – being a Web Service, Script or other embedded activity – to the input and output ports of the processor they are contained in. Data types are not explicitly dealt with in Taverna, i.e., processor port and activities parameters have no type assigned to them. Figure 4-3 below shows the structure of a Taverna dataflow.

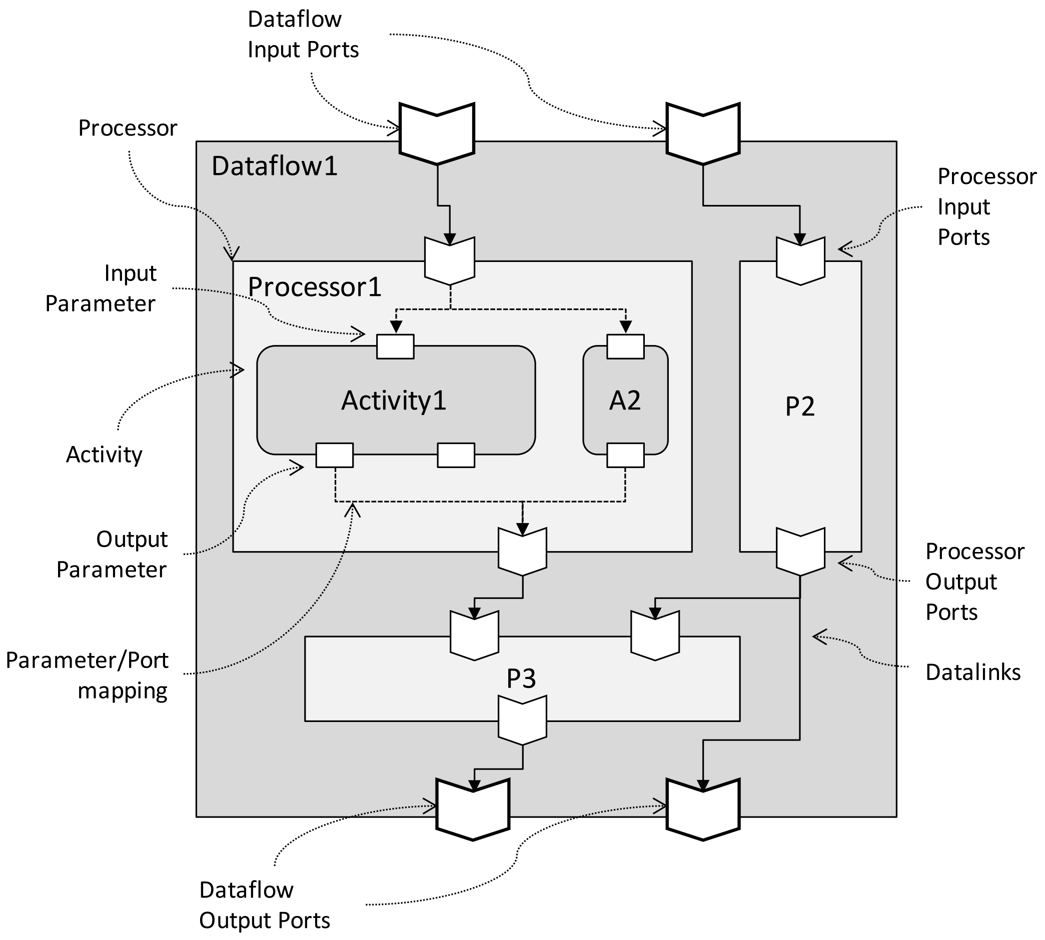


Figure 4-3. Structure of a Taverna workflow.

4.3.3. Analysis of the myExperiment Repository

In this section, we detail the steps taken to analyze the myExperiment workflows. The workflows are publicly available and can easily be obtained via Web Browser or automated HTTP fetching tools. We analyzed the workflow following the steps described in Section 4.2 to obtain the semantic information about the services. Some decisions and compromises where made in order to have a dataset that was comprehensible yet simple to handle in practice:

- Only Taverna 2 workflows were used;
- Workflows containing features that do not contribute to the problem of service composition – namely, workflows with nested dataflows and those with processors containing more than one activity – were not used.
- Control structures inside the workflows were not taken into account (e.g., ordering structures that define that a processor should be executed before another one, even though they do not have an input-output relationship, were ignored).
- Malformed workflows were excluded from the dataset (i.e., workflows with syntax errors or whose resulting composition is not valid according to Definition 4-5).

By using only Taverna 2, we end up with 570 workflows, from the initial 2000+ available in the repository. The number of "good" workflows – those that meet the conditions listed above – is 425. This final set of workflows was used in remaining of this Thesis.

Taverna workflows contain activities of various types – classes in Taverna terminology –, not only Web Services invocations. In this work, we opted for using all kinds of activities as if they are regular services, not only proper Web Services. Our goal was to be able to build a complete composition out of the pieces in the repository, and for that end, it was necessary to have these additional types of tasks available. In our experience dealing with Taverna workflows, we noted that these accessory tasks represent in fact a great part of real workflows, helping preprocessing and reformatting data to and from the Web Services (examples of activity classes that perform such tasks are XMLInputSplitter and XMLOutputSplitter). From the practical point of view of the semantic extraction process, the extra effort needed was minimal. Some extra care was necessary in order to ensure that one activity used in one workflow was the same (or not) used in another. The activity type, input and output parameters and extra information (depending on the activity type, for example, the WSDL URL for Web Services and database name for Biomart activities) were used to generate unique identifiers. In the end, the total number of services gathered from workflows was 1094, 211 of which were SOAP/WSDL Web Services (19%) and 22 were RESTFul Web Services (2%).

Although for the myExperiment ecosystem there was the possibility of including BioCatalogue services in the repository we opted to keep the set of services restricted to those gathered from workflows in order to have a simpler – also more constrained – scenario. The use of external service repositories to help the composition process is planned for the continuation of this work.

Figure 4-4 presents the histogram of the number of service invocations per workflow. This figure helps visualizing the size of the workflows in the database. The majority of workflows make ten or less service invocations, with the frequency decreasing consistently for higher number of invocations. Almost all workflows need less than 35 invocations to accomplish their tasks – 424 workflows or 99.8%. Only in one extreme case, 56 service invocations were used.

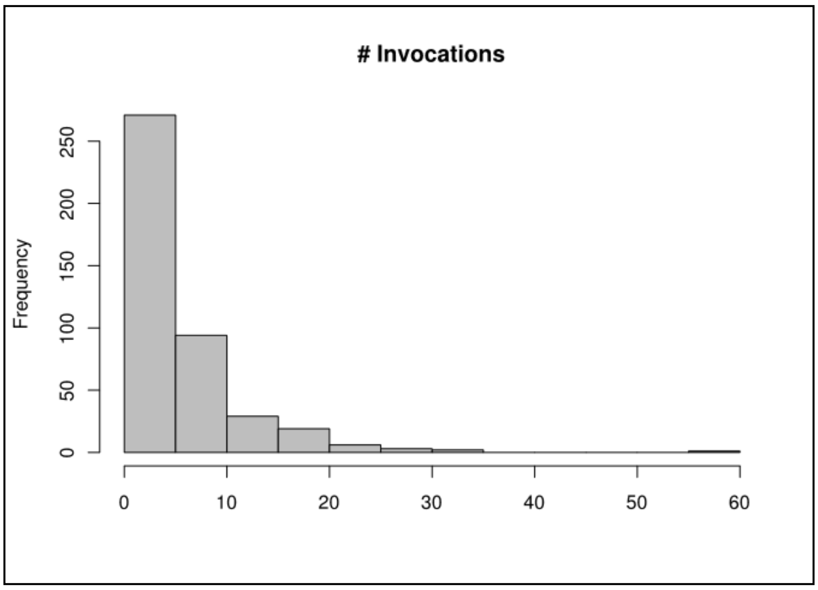


Figure 4-4. Histogram of the number of service invocations per workflows.

Looking at the Web Service usage histogram of Figure 4-5 it is possible to see the low reuse of services: 80% of them were employed in a single workflow. This level of service reuse is also observed in service repository as a whole, with only 23% of the services being used more than once.

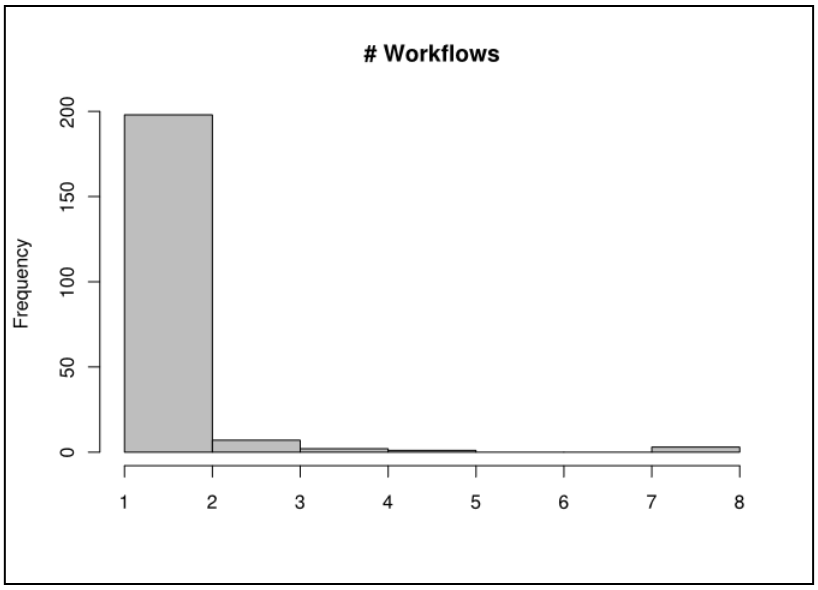


Figure 4-5. Histogram of the number of workflows that use each Web Service.

4.3.4. Reverse Engineering Results

4.3.4.1. Parameter Compatibility

The parameter analysis in the myExperiment repository has its own challenges due to the need of including all activity classes. This happens because most activity classes in Taverna have no proper parameter typing; only for Web Service activities, it is possible to obtain the parameter type by analyzing the corresponding WSDL description. Therefore, it was not possible to establish a hierarchy of types, as described in Section 4.2.1. One way to deal with this problem would be to assign a generic type – "string", for example – to all parameters with unknown type. This would allow any parameter interactions, i.e., outputs of an activity could be used as input to any other activity. Instead of using this permissive approach – which would make computation more expensive and solutions less reliable – we opted to have each parameter, either input or output, belonging to its own type. The parameter type, in this case, is determined by the parameter and the service identifiers, e.g., for service called "$SetBalance$" whose first parameter is "$AccountNumber$", the resulting type is identified as "$SetBalance\_AccountNumber$". For the parameters associated with Web Service invocations, the type used was that present in the WSDL description, however no attempt was made to relate WSDL-described types to each other based on their internal structures.

Using automatically generated parameter types, as described above, the relationships between types have the form "input-type is compatible to output-type". Since input parameters cannot be connected to other inputs, the same being true for outputs, relations of the form "input-type is compatible to input-type" and "output-type is compatible to output-type" did not exist. The resulting structure is better described as a Parameter Compatibility Matrix (PCM) instead of a type hierarchy, which relates all services' parameters and tells whether an output can be connected to an input. We extended this matrix to contain not only the binary relation ("compatible or not") but also the number of times the input and output parameters were connected in different workflows. This number can be used, for example, to rank the output parameters that match certain input based on how frequently the connection output-input was used by other developers, although in our experiments we were interested only in the binary relation. An example of PCM for a simple banking scenario (fictitious) is shown in Figure 4-6. In this example, the parameter of type *GetBalance_Balance* can be assigned to parameters of type *SetBalance_NewBalance* since in two compositions; the output *Balance* of service *Get_Balance* was connected to the input parameter *NewBalance* of service *SetBalance*. The same reasoning follows for the remaining cases in the matrix; blank cells are used to denote incompatible parameters, i.e., assignments that did not occur in the compositions.

Output Parameter Types / **Input Parameter Types**	GetBalance_AccountNumber	SetBalance_AccountNumber	SetBalance_NewBalance	GetAccountNumber_CPF	...
GetBalance_Balance			2		
SetBalance_Status					
GetAccountNumber_AccountNumber	1	3			
GetCPFbyName_CPF				4	
...					

Figure 4-6. Parameter Compatibility Matrix for a simple banking scenario.

After analyzing the set of compositions, we extracted a total of 1568 input and 1422 output parameters, what prevents us from presenting the complete compatibility matrix in a readable way. The resulting compatibility matrix is very sparse, with only 0.06% of the possible output-input assignments present (with 1404 unique assignments, i.e., disregarding repeated assignments). An input parameter is assigned in average 1.6 times, with one case counting 681 assignments (from 282 different services). Each output parameter is used 1.5 times on average, and 55 times in the extreme case (29 unique services).

4.3.4.2. Service Relationships

In order to allow the visualization of the service relationships we used a reduced set of 100 workflows (the first 100 from the dataset) to generate the graphs shown in this section. Some graph statistics were computed for both the reduced and the complete set of workflows using the graph visualization and analysis tool Gephi [7]. These statistics – e.g., node degree, graph diameter and average path – can reveal interesting facts about the service relations. For a complete reference on these graph measures and their meanings please refer to [51]. The first relationship analyzed was the service relation based on workflow usage, where services are connected if used together in one or more workflows. The graph for this relation (for 100 workflows) is shown in Figure 4-7.

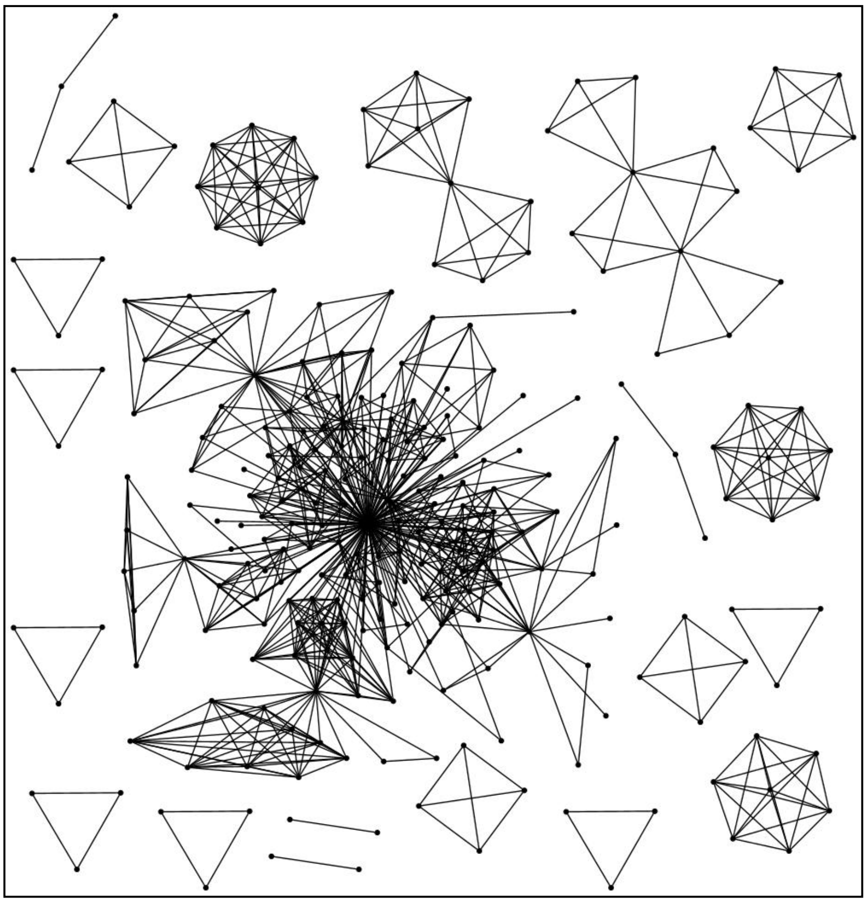


Figure 4-7. Service relation based on workflows for the 100-workflows dataset.

It can be seen that the graph is not strongly connected (21 component graphs were found) but contains a giant main component with 64% of the nodes. The same holds for the complete workflow set, which contains 82 components with one of them holding 66% of the graph. In Table 4-1 some graph metrics are shown for the 100- and 425-workflows datasets. Interestingly, note that the diameter of both graphs remains the same despite the difference in size, the same happens to the average path length. The low graph diameter together with high average cluster coefficient point to a *small world* graph in both cases (of course, since the graph is not connected the small world premise of low number of hops between any pair of nodes does not apply entirely). In both cases, there is also a hub node with very high degree – 137 and 610 for 100- and 425-workflows respectively – which in this case consists of a utility activity used frequently to model string constants in the Taverna workflows.

Table 4-1. Workflow-based Service Relation Metrics.

	100 workflows	**425 workflows**
Number of nodes	265	1052
Number of edges	810	4493
Average degree	6.1	8.5
Diameter	4	4
Average cluster coefficient	0,86	0,91
Giant component size	170 (64%)	705 (66%)
Number of components	21	82
Average path length	2.3	2.2

The resulting graph for the precedence relationship for 100 workflows is shown in Figure 4-8. In this graph, services are connected if one precedes the other one in any workflow analyzed. The graph is directed and links span from preceded services to predecessors.

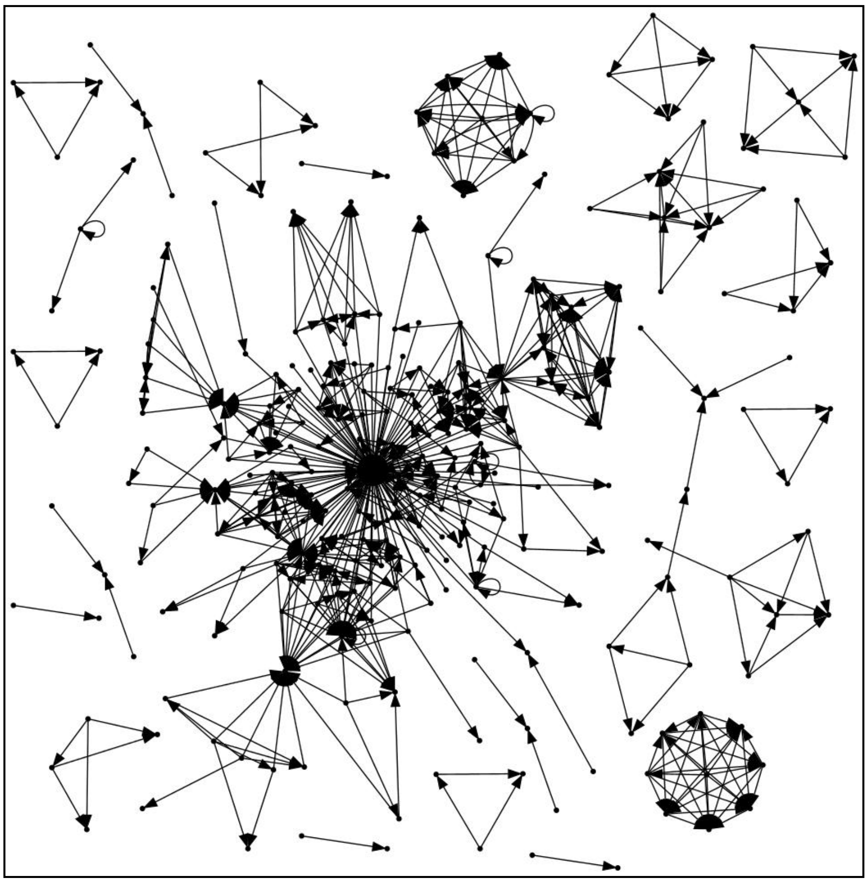


Figure 4-8. Service precedence relations for the 100-workflows base.

The graph in Figure 4-8 contains fewer nodes than the previous one since orphan nodes were not included. The overall pattern of the workflow-based relation is present in this graph, with a component carrying 65% of the nodes and several smaller components present. Compared to the previous relation, service precedence presents a larger network diameter and a slightly lower cluster coefficient. Other statistics are shown in Table 4-2 below.

Table 4-2. Service Precedence Relation Metrics

	100 workflows	425 workflows
Number of nodes	258	1035
Number of edges	625	3611
Average degree	3.4	5.4
Diameter	7	7
Average cluster coefficient	0.7	0.78
Giant component size	167 (65%)	670 (65%)
Number of components	20	78
Average path length	2.7	2.6

The service dependency relation (Figure 4-9) is the more restrictive one in terms of the service connections present compared to the two previous cases. Please recall that in this graph two services are connected only if one service uses the output of the other service in any composition. This graph helps to illustrate how often services are directly connected. The average degree in this case is close to 1, i.e., in general, the output of a service is useful to a single other service only. The statistics for this relationship are shown in Table 4-3.

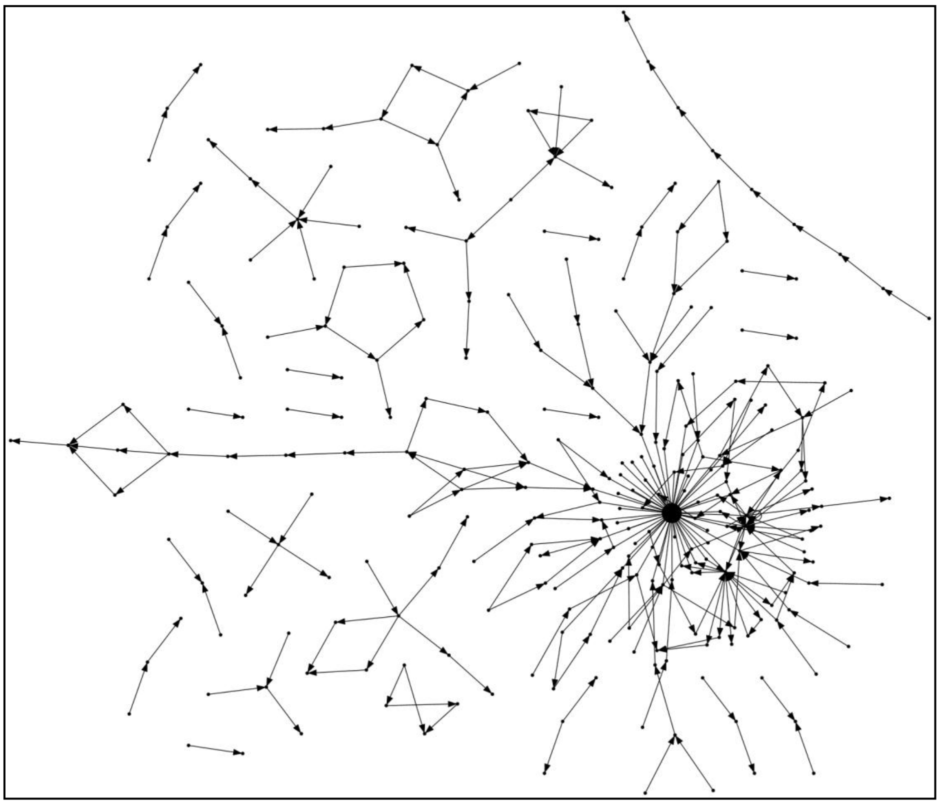


Figure 4-9. Service dependency relation for the 100-workflows base.

Table 4-3. Service Dependency Relation Metrics

	100 workflows	425 workflows
Number of nodes	262	1044
Number of edges	289	1210
Average degree	1.1	1.2
Diameter	16	18
Average cluster coefficient	0.08	0.06
Giant component size	157 (60%)	680 (65%)
Number of components	27	89
Average path length	4.2	4.6

4.4. Summary and Discussion

In this chapter, we presented a process for discovering semantic information from a set of existing workflows with the purpose of feeding an automatic service composition procedure. This application of this process is supported by the fact that formal semantic descriptions are not available for most services, and therefore alternative ways of gathering this information are necessary. We assume that a workflow database is available for that purpose. A reverse engineering process is applied to this database in order to extract meaningful relations between services and their inputs and outputs. In our case, we used the myExperiment repository of publicly available scientific workflows. However, an organization could apply these techniques to their private business process library without major changes.

This chapter was inspired and has similarities with the work presented by Tan et al. in [80]. In that paper, service relationships were established based on the occurrence of services in the workflow database. Their investigation focused on extracting these relationships and analyzing them from the perspective of social networks, aiming at obtaining interesting knowledge about the workflows. One of their findings – which we also identified in Section 4.3.3 – is that service reuse is low in the myExperiment database, i.e., only few utility services are used frequently while most data processing Web Services are used in only one workflow. In their follow-up works, the authors used the service relation graphs to provide to workflow developers GPS-like routing ("which services connect these two other services?") [79], and service recommendations (e.g., "given the services the developer has already used in the workflow, which one he/she is more liked to need?") [96]. For that latter purpose, workflow authorship and group membership, also gathered from the myExperiment website, were used to help filtering the services to be recommended. In this Thesis, however, we were interested in building relationships that could be used in automated composition. We focused on the structure of service and parameter relationships instead of looking at other external sources of information about the service (like authorship) since they often are not available or reliable.

The work in [55] also used the idea of extracting semantic information – parameter type information, specifically – from WSDL files for the purpose of automated service composition. In that work, however, its authors did not have a repository of compositions to extract additional information on the services relationships. In [92], [83] and [53], the OWL-S descriptions of services were reverse-engineered and abstract "methods" telling how to use the services derived in the process. These works use the service's Process Model, which provides an abstract description of how clients have to interact with a multi-step service (i.e., a service that can be partially invoked in several steps). Differently from them, we adopt a single step, request-response service model (equivalent to Atomic Processes in OWL-S and a WSDL operation), and extract information on how various services interact with each other from concrete compositions.

The resulting structures of the analysis of the myExperiment workflows were stored in an internal representation suitable to be used with composition algorithms. At this point no semantic description of the services – using, e.g., OWL-S – was generated. The generation of such descriptions is a subject of future work. In the next chapter, we describe the automated composition algorithms developed in this Thesis and how it uses the information gathered from a workflow repository to generate new workflows.

Chapter 5 Proposed Algorithms

The problem of composing services is very similar to the problem of finding a suitable plan to execute a task: starting from some given initial state (the known inputs), find a sequence of actions (services) that achieve the desired goal (provide the expected outputs). This perfect match explains why automated AI planning is the preferred technique for addressing the problem of automatically composing services, as seen in Chapter 3, and it was the path chosen in this Thesis. In Section 5.1 we adjust the terminology and map the concepts from classical planning (as seen in Section 2.3) to the ones in service composition. Once the concepts are clarified, we describe the algorithms used and introduced in this Thesis. We begin with the base algorithm taken from the AI planning literature, Graphplan, and explain what makes it suitable for the task at hand. Then we propose improvements to this base algorithm in order to obtain service compositions more similar to those created by human developers. Another extension we present to the algorithm adds support to incomplete initial specifications, a feature designed to help the developers in the task of creating new services even if they do not have complete knowledge on their outline beforehand. These new features have their cost in the algorithm performance, hence a more sophisticated approach to find the compositions is necessary. An extended algorithm, based on another well-known planner, is proposed to cope with this problem. The chapter ends with a summary and discussion.

5.1. From Planning to Service Composition

In this Thesis we are working with Classical Planning (or STRIPS planning) with extensions. As seen in Chapter 2 (Background), a planning task is comprised of problem and domain descriptions. The planning problem defines the state the system should hold before and after the execution of the resulting plan. The planning domain describes, among other things, the actions available for solving the problems in terms of their observable external behavior. Each action has a set of preconditions that must hold for it to be applicable and a series of effects in the state of the systems. In Classical Planning, effects and preconditions are seen as logical formulas over literals or propositions. Actions have both positive effects (propositions that are made true) and negative effects (propositions made false). Negative effects may cause actions to be incompatible (or be mutually exclusive) in a given state, e.g., if the effect of one action removes the precondition of another. Plans are comprised of actions, usually ordered, although a complete ordering is not always required. Valid plans are those where, for all actions in the plan, the preconditions hold if the order of execution of the plan is followed. A plan is a solution for a planning problem if it is valid and accepts the initial state of the problem (the plan's precondition holds in the initial state) and the state brought by executing the plan is a valid final state of the problem.

In the SOA realm, compositions are formed by wiring services together. Services are usually described with just enough information to allow them to be properly used, which means that only their inputs and outputs parameters are described. Sometimes, extended information in the form of formal semantics is present. Automatically composing services, in this context, means finding a combination of services that can be connected together while ensuring that each one is provided with the information it needs, i.e., all its input parameters are correctly assigned.

As for services, actions can also accept parameters: objects associated with the problem domain are passed as arguments. However, the set of propositions, not the availability of objects, determines when an action is applicable or not. In service composition, as modeled in this Thesis, services are applicable once the information they need is available, that is, there are instances of parameters (that result from invoking a service) that match their input parameters. The state of the system is determined by the parameter instances available, and therefore the precondition of a service is based on its input parameters. The service precondition takes into account the types of the input parameters, including compatible types. For example, consider a service $SetBalance$ that has input parameters $AccountNumber$ and $Balance$. $AccountNumber$ is compatible only with $String$ parameters, while $Balance$ accepts both $Integer$ or $Real$ values. The precondition for service $SetBalance$ will be:

$$Pre(SetBalance) = available(String) \wedge \big(available(Real) \vee available(Integer)\big)$$

The predicate $available(t)$ states that an instance of type t exists. Let us assume that $SetBalance$ returns a Boolean with the status of the operation. The effect of the service is given by:

$$Effect(SetBalance) = available(Boolean)$$

The effect means that a boolean parameter is available to be used by other services. Parameter types such as $Real$, $Integer$ and $Boolean$ can be seen here as domain objects.

In this Thesis, we follow the natural path of mapping services to actions and compositions to plans in order to use planning techniques for the purpose of automated composition. Table 5-1 summarizes the mapping between planning and service composition concepts.

Table 5-1. Mapping between planning and service composition concepts.

Classic Planning	**Service Composition**
Domain Description	Service and Type Repository
Problem Description	Composition Specification
Plan	Composition
Action	Service
Preconditions	Input Parameters
Positive Effects	Output Parameters
Negative Effects	N/A
Proposition	Parameter

An important difference from classical planning that arises from the mapping above is that services cannot make existing instances disappear (at least this is not standard behavior in BPEL and other composition languages, once a variable is instantiated it is available to all subsequent service invocations). Therefore, there are no negative effects on invoking a service, i.e., no parameter instance ceases to exist because of its invocation, only new instances will be made available to other services. If no other constraints are associated with the services (e.g., via semantic annotation) that restricts the way services are combined then no mutual exclusion relations will exist, making the problem easier to solve. In our case, additional constraints are added in order to improve the quality of the solutions, as will be discussed in the following sections.

5.2. Base Algorithm: Graphplan

As seen in Chapter 2, there are many ways for modeling the planning problem – e.g., state space, planning space, propositional satisfiability – and for each one several algorithms exist. We chose to work with an algorithm that is known as being fast and that fits well with the properties of the Service Composition problem: Graphplan. In this section, we describe the general Graphplan algorithm while in Section 5.3 we explain why it suits the problem at hand. Graphplan, proposed by Blum and Furst [11], works by building a planning graph of a relaxed version of the planning problem and then attempting to extract a valid plan from the planning graph. If no valid plan is found, the planning graph is expanded further and the process is repeated until a valid solution is found. The Graphplan algorithm is shown in Algorithm 5-1:

```
Input: Initial state $s_0$ and final state $g$
Output: Valid solution $plan$
G = new graph with $s_0$ at layer 0
for $k = 1, 2, 3, \ldots$ do
    expand($G$, $k$)
    if expansion failed then
        return FAIL
    end
    if $g$ found in layer $k$ of $G$ then
        $plan$ = extract($G$, $g$, $k$)
        if success then
            return $plan$
        end
    end
end
return FAIL
```

Algorithm 5-1. Graphplan algorithm.

A planning graph is a layered graph with alternating layers of propositions and actions. The top proposition layer contains the initial state of the system. The next layer is comprised of actions whose preconditions are present in the top layer. The effects of these actions are then added to the next proposition layer, which provide the preconditions for the next action layer, and so forth. Formally, let $G = \langle s_0, A_1, P_1, A_2, P_2, \ldots, A_k, P_k, \ldots, A_m, P_m \rangle$ be a planning graph, with s_0 being the problem's initial state. Each level k of G is comprised by a layer of actions A_k followed by a layer of propositions P_k, with $P_0 = s_0$. The extracted plan Π has the same layered structure of the planning graph: $\Pi = \langle s_0, \pi_1, \rho_1, \pi_2, \rho_2, \ldots, \pi_n, \rho_n \rangle$; π_j and ρ_j, $j = 1,2,\ldots,n$, are the action and proposition layers of the plan respectively.

In our Service Composition analogy, the layers are comprised of service invocations and the parameters instances associated to them. Figure 5-1 shows an example of a planning graph, generated from a very small sample of our working database of composition. It is possible to see the layers of service invocations (record-shaped nodes), the top part of each service node being its input parameters and the bottom part the output parameters. Parameter instances are represented implicitly by the assignments (arcs) from one service invocation's output to the input of another service invocation. Grey nodes are special NO-OP (no operation) invocations used to pass parameters from one layer to the next one, making them available to the next layer of services. The box-shaped nodes at the top and bottom of the graph are the input and output parameters, respectively, specified for the composition.

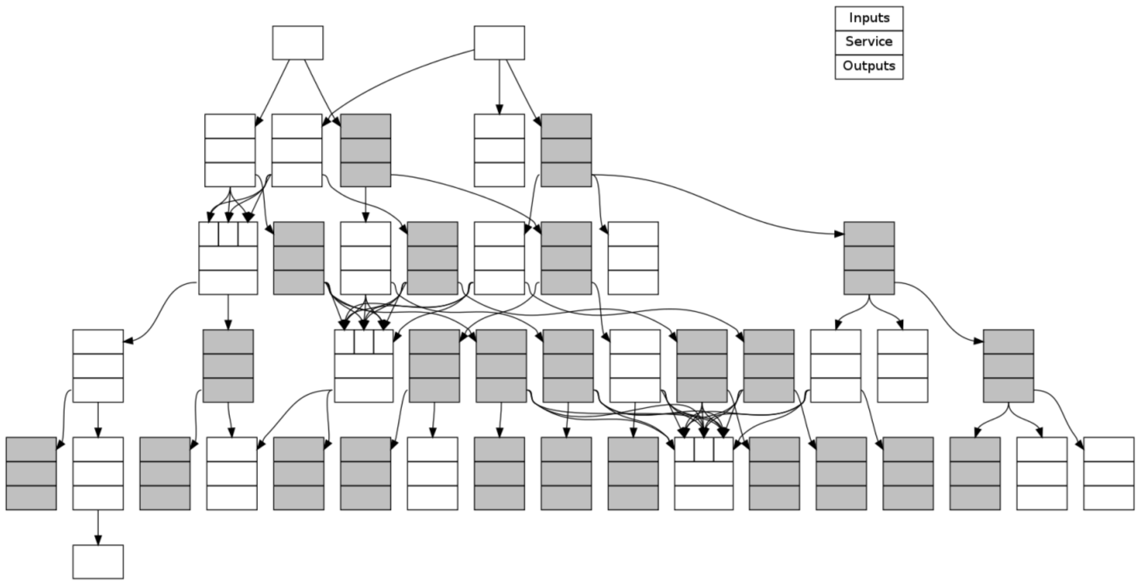


Figure 5-1. Example of planning graph.

Graphplan relaxes (i.e., simplifies) the planning problem during graph expansion by ignoring the negative effects of actions. Therefore, an action is applicable in a given layer even if another action that denies its preconditions exists. The Graphplan algorithm, however, keeps track of these conflicts, maintaining a pair-wise list of mutual exclusion relations between actions and propositions. Two actions in the same level are mutually exclusive if they have inconsistent effects – the effect of one action negates the effect of the other –, if one negates the precondition of the other, or if they have conflicting preconditions. Two propositions are mutually exclusive if one negates the other or there is no mutex-free pair of actions that provide them. Actions with conflicting preconditions (i.e., mutex in the previous layer) are not included in the graph during expansion. The expansion algorithm is shown in Algorithm 5-2.

```
Input: Graph $G$ and layer $k$
Output: Expanded $G$
1 $A =$ actions whose preconditions match $P_{k-1}$
2 if $A = \emptyset$ then return $FAIL$
3 Exclude from $A$ any action with mutex preconditions
4 Make action layer $k$ of $G$ $A_k = A$
5 Make $P_k =$ positive effects of actions in $A_k$
6 Update mutex relations for level $k$
```

Algorithm 5-2. Graph expansion algorithm.

The graph is expanded until the bottom layer contains the goals of the planning problem. At this point, the algorithm tries to extract a valid plan searching upwards for combinations of actions that provide the end goals. The preconditions of the actions found become the new goals, and the search continues with the layer above until the top layer, that contains the initial state, is reached. The search can eventually fail at some layer k: all combination of actions are mutually exclusive, and therefore cannot be applied together. When this happens, the algorithm backtracks to the last successful layer and tries another combination, until a solution is found or no more combinations exist. In order to prune the search space and speed up planning extraction a table of "bad" proposition combinations per layer k is kept – the $nogood(k)$ table. A proposition set g' that yields failure is added to $nogood(k)$ and if the algorithm happens to find g' at level k again it prunes the search immediately. The Graphplan algorithm continues expanding and searching the planning graph until a solution is found or no progress is made (no new propositions or actions are added), at which point the algorithm fails completely. The plan extraction algorithm is shown in Algorithm 5-3:

```
Input: Graph $G$, goal propositions $g$ and layer $k$
if $k = 0$ then return empty plan; // Success
$A$ = all sets of non-mutex actions in layer $k$ of $G$ that provide $g$
if $A = \emptyset$ then return FAIL
foreach set of actions in $A$ do
    $g'$ =preconditions(actions)
    if $g' \notin nogood(k)$ then
        plan =extract($G, g', k-1$)
        if success then
            Add actions to level $k$ of plan
            return plan
        else
            Add $g'$ to $nogood(k)$
        end
    end
end
return FAIL
```

Algorithm 5-3. Plan extraction algorithm.

The expansion of the planning graph is polynomial both in time and in space with the size of the planning problem (number of actions and propositions involved). The most computationally demanding part of the algorithm is the plan extraction, which searches the state space provided by the planning graph for a valid plan. The overall complexity of the algorithm is the same for classic planning: PSPACE-complete [25][11]. The algorithm terminates and is both sound (i.e., the plan returned, if any, is a solution of the problem) and complete (if the problem has a solution, it will eventually find it) [25].

5.3. GRAPHPLAN FOR SERVICE COMPOSITION

The Graphplan algorithm suits well the Service Composition problem since the problem, in its basic form, is already "relaxed": because services have no negative effects, no mutual exclusion happens between them. One consequence is that the search algorithm will not fail to build a plan once the expansion reached a layer where all intended outputs are present. The relaxed version of Graphplan becomes polynomial to the total number of actions, or services in our case [34]. We use this version of the algorithm in the experiments, comparing it against state-of-the-art planners (Chapter 6). We also made some improvements to the algorithm in order to obtain "better" solutions, as will be discussed in the following sections. These changes, however, reclaim the original complexity of Graphplan.

5.3.1. Enforcing Input Parameters

General-purpose planners try to find plans with minimum number of actions. Similarly, the Graphplan algorithm finds plans with minimum execution depth, i.e., the solution found should have the least layers possible. It is reasonable to assume that a minimum or faster plan is preferred in the general case. However, sometimes this leads to plans that do not make use of the all the information provided as initial state. Again, in classic planning this is not a problem. However, once we map propositions to parameters, this means that compositions that ignore some of the initial input parameters can be returned as solutions. If we have in the service repository services that need no inputs (for example, services used to model string constants), it is not uncommon for the planners to generate compositions that use none or just a few of the inputs provided. Although not strictly incorrect, these compositions may not be exactly what the developer intended.

In order to give more control to the developer, we propose an extra configuration option to the composition specification: the maximum number of unused inputs, $\Delta_{max} \geq 0$. This number tells how many of the provided input parameters can be left unused by the resulting composition. If the number is 0 (zero), then all inputs must be present in the final solution; if it equals or is greater than the number of inputs then we have the original behavior of the (relaxed) Graphplan algorithm.

We implemented this important feature by changing the search procedure of the Graphplan algorithm to account for the number of used inputs. If the search procedure reaches the top layer, it checks the used inputs and, if it meets the specified configuration, it then returns with success, otherwise backtracks and another search round is made with another set of services. The general behavior of the algorithm follows that of Graphplan with mutual exclusion. In fact, we can model this feature as a type of mutual exclusion involving several actions. For example, we can keep track during graph expansion of the maximum number of composition inputs reachable from a given service invocation I, $used_{max}(I)$. When searching for a solution, a set of services of a given layer will be mutually exclusive if the sum of their maximum used inputs $\sum_j used_{max}(I_j)$ is less than the total number of inputs minus Δ_{max}. The implementation of the algorithm, however, does not keep track of the maximum number of used inputs since, in early tests, the time gained by pruning the search space was consumed by the overhead of maintaining this counter for all combinations of invocations, per level. The $nogood$ table of bad states, as proposed in Graphplan, is still used to prune the search space, though. The modified search algorithm is shown in Algorithm 5-4:

```
   Input: Graph $G$, goal propositions $g$ and layer $k$
1  if $k = 0$ then
2      if $|s_0 \setminus g| \leq \Delta_{max}$ then
3          return empty plan // Success
4      else
5          return FAIL
6      end
7  end
8  $A$ = all sets of actions in layer $k$ of $G$ that provide $g$
9  if $A = \emptyset$ then return FAIL
10 foreach set of actions in $A$ do
11     $g'$ =preconditions(actions)
12     if $g' \notin nogood(k)$ then
13         plan =extract($G, g', k-1$)
14         if success then
15             Add actions to level $k$ of plan
16             return plan
17         else
18             Add $g'$ to $nogood(k)$
19         end
20     end
21 end
22 return FAIL
```

Algorithm 5-4. Plan extraction with unused inputs checking.

The difference from the original extraction procedure is in line 2, where we check if the current goals differ in at least Δ_{max} from the set of original goals once we reach level 0 of the planning graph. In other words, we check if the plan uses enough input parameters according to the Δ_{max} setting.

5.3.2. Incomplete Input Specification

Another fundamental feature added to the base algorithm was the ability to deal with incomplete information, specifically, the lack of some of the input parameters necessary to find the solution. Automated service composition algorithms – all those reviewed in Chapter 3 at least – fail to provide a solution if the specification is incomplete, leaving to the developer the task of finding out which new parameters should be added in order to obtain a valid composition. The present feature was designed to streamline the composition development, helping the developer finding the composition it wants even if it does not know all input parameters beforehand.

The developer can specify the number of missing or "extra" inputs – in addition to those it knows – the algorithm is allowed to accept in the composition. It will try to find such a solution that meets this constraint. This implies allowing services that have not all their input parameters covered to be part of the planning graph during the expansion phase of the Graphplan algorithm. Let say the $\Gamma_{max} \geq 0$ is the maximum number of missing parameters. During graph expansion, those services that match the available parameter instances of the last layer of the graph are looked up to form the new layer. For any given service, if all but at most Γ_{max} of its input parameters match those available in the last layer then it can be added to the next layer. Since the limit for extra parameters in the final plan is Γ_{max}, it will not be helpful to add services that have more than Γ_{max} input parameters not covered.

Simply imposing the limit of extra parameters during expansion does not guarantee that a valid solution exists. The solution extraction phase must keep track of the number of extra parameters in the current plan and prune the search if the number exceeds the limit Γ_{max}. Here we have a situation akin to the one found with Enforced Inputs, where sets of invocations in a given layer are mutually exclusive if the number of unmatched parameters they required is above Γ_{max}. Once this occurs, the extraction algorithm backtracks and tries another combination of invocations, until one combination is found – success – or all combinations were explored – failure. The modified extraction algorithm is presented in Algorithm 5-5.

```
Input: Graph $G$, goal propositions $g$, missing goals $\gamma$, layer $k$
1  if $|\gamma| > \Gamma_{max}$ then return $FAIL$
2  if $k = 0$ then
3      if $|s_0 - g| \leq \Delta_{max}$ then
4          return $empty\ plan$ // Success
5      else
6          return $FAIL$
7      end
8  end
9  $A$ = all sets of actions in layer $k$ of $G$ that provide $g$
10 if $A = \emptyset$ then return $FAIL$
11 foreach $set\ of\ actions\ in\ A$ do
12     $g'$ = preconditions($actions$)
13     if $g' \notin nogood(k)$ then
14         $\gamma' = \gamma \cup (g' \setminus P_{k-1})$
15         $plan$ =extract($G, g', \gamma', k-1$)
16         if $success$ then
17             Add $actions$ to level $k$ of $plan$
18             return $plan$
19         else
20             Add $g'$ to $nogood(k)$
21         end
22     end
23 end
24 return $FAIL$
```

Algorithm 5-5. Plan extraction with verification of unused and missing inputs.

Line 1 of the algorithm checks if the number of missing inputs, $|\gamma|$, exceeds the limit Γ_{max}, in which case the search is aborted and another set of service invocations is tried. The algorithm keeps track of missing inputs in line 13, where the new set of missing inputs γ' adds to the old set γ the service inputs not found in the level above.

The consequence of allowing services with partial input matches into the planning graph is a potentially much larger search space for the solution extraction phase. When combined with the Enforced Inputs feature, the time necessary to find a solution (or to return failure) can be prohibitive. To work around this problem, we explored a heuristic solution to the problem of finding a composition with both enforced and extra input parameters.

5.4. Fast Forward with Enforced and Missing Inputs

While aiming at improving planning performance, we developed an alternative implementation of the proposed features using another classic planning algorithm: The Fast-Forward (FF) algorithm, proposed by Hoffman and Nebel [34]. The FF is a planning heuristic that performs a state-space search using the relaxed Graphplan algorithm to compute a heuristic value for each state. Each state in the FF algorithm is a sequence of actions that yield the set of propositions that pertain to the state. Positive and negative effects of the actions are taken into account when building the state. From the set of literals of the current state a relaxed Graphplan execution is performed, resulting in a relaxed solution for the planning problem (or a failure). The relaxed solution is used to measure how close to the final goal is the current state. In this case, the value associated with the state will be the number of actions present in the relaxed solution. This value is used to guide the search across the state space.

Given a state, the set of next states is computed using the concept of *helpful actions*. The helpful actions of a state are the set of actions from the first layer of the planning graph of that state – i.e., the set of actions applicable from the propositions of that state – that contribute to the next layer of the graph (i.e., add new information). The idea of helpful actions is to prune the search space while keeping useful actions at hand. The FF algorithm employs other strategies in order to speed up the search – for a complete reference see [34].

Our algorithm is inspired in the FF in the sense that it also uses Graphplan to guide the search. However, while FF uses the relaxed solution only as a measure of the state value, we use the relaxed solution as the final solution itself. In our case, a relaxed solution is one that potentially violates the constraints imposed by the developer on the number of enforced (Δ_{max}) and extra inputs (Γ_{max}). Once it is determined that a relaxed solution does not violate these constraints (what can be done quickly by traversing the resulting plan once), there is no reason to discard it. In Figure 5-2 it is shown how simply reordering the invocations in an expansion of the planning graph can provide a valid solution. In this simple example, we are interested in outputs of type B, C and D, and provide a single input of type A. The constraint imposed is that at most one missing parameter might exist in the solution, $\Gamma_{max} = 1$. The solution found in (a) provides the target outputs but needs two extra parameters to match inputs Y and X (Services 1 and 3 respectively). The solution in (b) meets the constraint of one single missing input by postponing the invocation of Service 1 so that the output of Service 2 can be assigned to input Y.

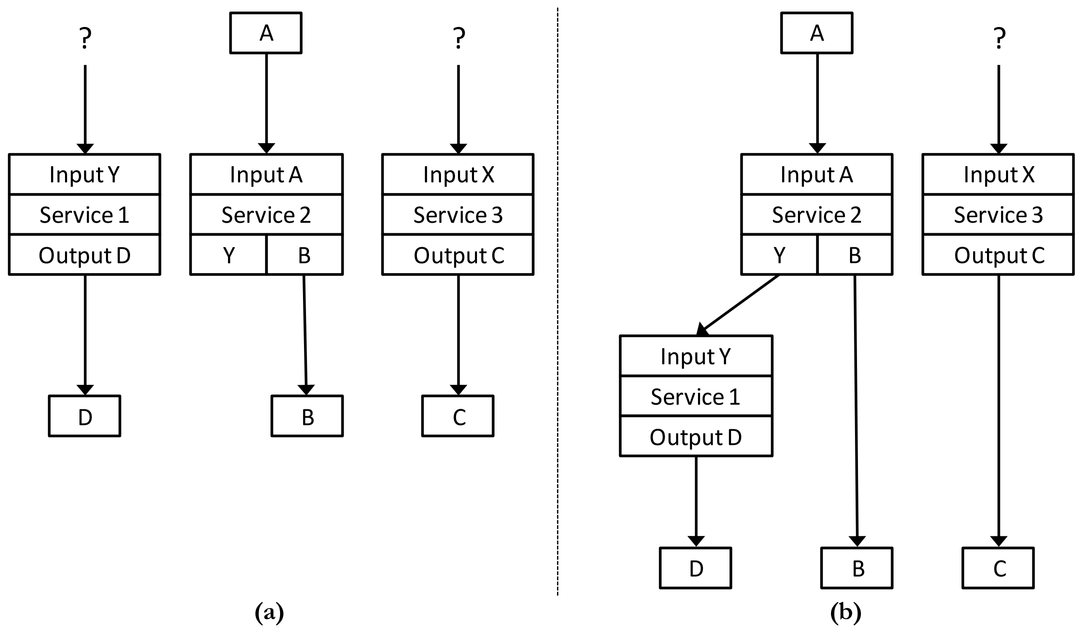


Figure 5-2. Example of graph expansions: a) not meeting $\Gamma_{max} = 1$ b) meeting $\Gamma_{max} = 1$. (NO-OP invocations are not shown).

In our algorithm, each state can be seen as a different expansion of the planning graph. From each expansion, a relaxed solution is computed, and if is valid (i.e., meets the constraints) it is returned as the final solution. The solution is the first one found by the search algorithm, which does not backtrack in this setup. If it is not a valid solution, the heuristic value of the state, $h(S)$, is calculated based on amount by which the solution violates the constraints. In case no solution for the state exists at all then the state is discarded (i.e., the search does not continue from it). The first state consists of the basic relaxed expansion of the Graphplan algorithm from the global initial state (with the inputs provided for the composition). A relaxed search is performed in the planning graph and if the solution is valid, it is returned immediately. If no relaxed solution exists, then the algorithm fails as a whole. The next states will be based on the services immediately applicable from the initial state, similarly to the original FF. Each new state inherits the top portion of planning graph from its parent state (the state was derived from), adds one of the applicable services to the next level (and the NO-OPs necessary for propagating other parameters), and continues the expansion from that point on. Assuming each state has a depth (the state-wise distance from the initial state), a state and its parent share the same planning graph from depth 0 to the depth of the parent, and diverge starting from the depth of the state. Formally, a state S can be defined as $S = \langle S_p, G_S, \Pi_S \rangle$, where $S_p$is the parent state, G_S and Π_S are the planning graph and plan associated to S, respectively.

From a given state, the set of candidate next states is given by the applicable services from that state. The choice of this set determines both the accuracy and the length of the search. If all applicable services are selected, we have a higher probability of finding a solution but at the expense of a much longer computation time. Conversely, choosing too few services will provide quick responses but more frequent false-negatives (i.e., the algorithm returns failure when there is at least one valid solution). In our specific case, the number of applicable services is potentially large since services with incomplete input matches can be used. We chose to use in the algorithm as the set of applicable services, for a given state, all the fully-matched services (all input parameters assigned) of the next level of the planning graph for that state, along with the set of partially-matched services (some inputs missing) from the state's solution at the same level. The goal is to avoid having too many candidate services without pruning out all the partially matched services, which are expected as part of the final solution. The helpful actions set of state S, $helpful(S)$ is described formaly as:

$$helpfull(S) = fullyMatched \cup partiallyMatched \tag{5-1}$$

Where:

$$fullyMatched = \{a \in A^S_{depth(S)+1} : preconditions(a) \subseteq P^S_{depth(S)}\} \tag{5-2}$$

$$partiallyMatched = \{a \in \pi^S_{depth(S)+1} : \exists x \in preconditions(a), x \notin \rho^S_{depth(S)}\} \tag{5-3}$$

The sets A^S_k and π^s_k refer to the actions (service invocations) at level k of the planning graph and plan solutions associated with the state S; P^S_k and ρ^S_k follow analogously for the propositions (available parameters).

When the next state S is created, it inherits the planning graph of its parent state S_p up to the depth of S_p and adds up the corresponding invocation from $helpful(S_p)$. Before expanding the graph further, the number of missing inputs of the graph, $\Gamma(G_S)$, is computed. The expansion of the graph will take into account this number and allow partially matched services with at most $\Gamma_{max} - \Gamma(G_S)$ missing inputs.

The heuristic value of a state, $h(S)$, is used to measure how close to the goal is the state being evaluated. The FF algorithm uses the number of actions in the state's relaxed plan as its heuristic value. A value of 0 indicates that no actions are needed because the state is already a goal state (i.e., contains the goal propositions). In our variation of the algorithm, we are interested in the constraints that determine if a solution is acceptable or not: Δ_{max} and Γ_{max}. If no plan could be computed to state S, its value will be ∞ (or an arbitrarily large number), and the state is discarded. If a plan exists, the value of S depends on the number of unused, $\Delta(\Pi_S)$, and missing inputs, $\Gamma(\Pi_S)$, of the plan associated to S. In order to help reducing the search space, the number of helpful actions of a state (i.e., the number of successor states that can be derived from it) is also taken into account: the smaller the set, the better since it determines the branching factor of the search. The heuristic value of valid states is given by equation (5-4):

$$h(S) = \big(\Delta(\Pi_S) + \Gamma(\Pi_S)\big) \times |helpful(S)| \tag{5-4}$$

The original FF algorithm employs its own version of hill climbing, called Enforced Hill Climbing, to search the state space. In the original hill climbing algorithm, the search moves to a neighboring state S_{next} if the state is better – according to the heuristic value – than the current state S_{curr}, and proceeds until no better neighboring state is found. The last current state will be the result of the search. The enforced version proposed in [34] does a breadth-first search from the current state S_{curr} to find a strictly better (i.e., not equal) state S_{next} with $h(S_{next}) > h(S_{curr})$. The breadth-first search can go several levels deep – and several actions apart from the current state – until a better S_{next} is found. Once found, S_{next} becomes the current state and the actions from the former S_{curr} to S_{next} are added definitely at the end of the plan (there is no backtrack as this point). The search terminates with success when the current state is a goal state; it fails if no states strictly better than the current one can be found.

We opted to use a modified best-first search used in [13] instead of the Enforced Hill Climbing employed by the FF algorithm. In a best-first search (such as the classic A* algorithm), all the next states of the current state are evaluated, their heuristic values computed and put into a priority queue while the current state is taken out of the queue. The next state will be the one with the best heuristic value at the top of the queue. The modified version of the search traverses the state space by "jumping" immediately to a new state if its heuristic value is better than the current state, while keeping the current state at the queue if it was not completely explored. By doing this, the modified search attempts to evaluate fewer states, what is an interesting feature for our algorithm since every state evaluation implies a Graphplan execution comprised of expansion of the graph and the extraction of a solution. The search algorithm is shown in Algorithm 5-6:

```
   Input: Initial state $S_0$
   Output: Final state $S_{curr}$
 1 $queue$ = empty priority queue
 2 put $S_0$ in $queue$
 3 while $queue$ is not empty do
 4     $S_{curr}$ = head of $queue$ // do not remove...
 5     if $S_{curr}$ is goal then
 6         return $S_{curr}$
 7     end
 8     if hasChildren($S_{curr}$) then
 9         $S_{new}$ = nextState($S_{curr}$)
10         if $S_{new}$ has not failed then
11             put $S_{new}$ in $queue$ // becomes head if better
12         end
13     else
14         remove $S_{curr}$ from $queue$ // nothing else to do with $S_{curr}$
15     end
16 end
17 return $FAIL$
```

Algorithm 5-6. Modified Best-First search algorithm.

The function $hasChildren(S)$ (line 8) tells whether the state was already completely explored or not; $nextState(S)$ returns the next non-explored state directly derived from S (based on the helpful services associated with the state). States are kept in the priority queue according to their $h(S)$ value. Failed states are not put into the queue.

As the original FF algorithm, our implementation is both sound and incomplete [34]. Soundness is ensured since solutions are checked explicitly when determining whether the current state is a goal state or not. As with FF, the set of helpful actions may not include some actions (or service invocations) needed to find a valid solution, thus making the algorithm incomplete. Completeness is also hindered in our algorithm by the fact that, even if a state presents a "good" expansion (one that contains a valid solution), it is not guaranteed that the extraction algorithm will find it. Since the relaxed Graphplan is computed polynomially, the complexity of the algorithm will be determined by the search procedure. The Modified Best-First search performs a complete search over the state space in the worst case [13], being therefore PSPACE-complete with respect to classic planning problems.

5.5. Preferred Services with Reverse Graphplan

In order to provide guidelines to our algorithms whenever they have to choose between services we devised a process to find a set of Preferred Services. To compute this set, we apply the relaxed Graphplan expansion algorithm both in the regular direction (from the inputs to the outputs) and in the reverse direction (from outputs to inputs). The resulting planning graph for each phase is "reduced", a process that removes unnecessary services. The set of preferred services is given by the union of the services comprising both reduced planning graphs, the direct and the reverse.

As explained earlier, the original Graphplan does not guarantee that all inputs provided will pertain to the final solution, although the solution must provide all outputs. Conversely, applying the Graphplan expansion in the inverse direction (swapping services preconditions and effects) we have a planning graph where all inputs are reachable but not necessarily all outputs are used. The intuition is that combining the services in these graphs would provide the set of services likely to be part of the final solution. Before merging the graphs' services, though, we remove from them the services that are not in the path between an input and an output. This reduction algorithm simply removes the invocations at the last level whose outputs (or inputs, in the reverse case) are not outputs of the composition (inputs, respectively). It proceeds upwards, removing invocations whose outputs are not used in the level below, until it reaches the top level. Note that, in general, the reduced graph is not a valid solution: one input of an invocation may receive several assignments from different outputs, what violates our definition of a valid composition (see Section 4.1, item 4). The reduced graph, however, contains potentially various solutions. Its algorithm is presented in Algorithm 5-7:

```
**Input**: Graph $G$ and layer $k$
**Output**: Expanded $G$
1 $A$ = actions whose preconditions match $P_{k-1}$
2 **if** $A = \emptyset$ **then return** $FAIL$
3 Exclude from $A$ any action with mutex preconditions
4 Make action layer $k$ of $G$ $A_k = A$
5 Make $P_k$ = positive effects of actions in $A_k$
6 Update mutex relations for level $k$
```

Algorithm 5-7. Planning graph reduction algorithm.

The set of preferred services is used in our Graphplan variant during the solution extraction phase, which in hard cases is the more time-consuming phase of the algorithm (also true for the original Graphplan). In this phase, the algorithm has to choose between invocations at a given level that provide the same input to an invocation in the level below. The algorithm then uses the set of preferred services to guide this choice, ranking the candidate invocations depending on whether the associated service is preferred or not. In our FF implementation, we use the preferred services both in its Graphplan phase and for pruning the invocations that comprise the helpful actions set of each state. Since computing the set of preferred services requires two Graphplan expansions and reductions, plain versions of the algorithms were also evaluated.

5.6. Summary and Discussion

In this chapter, we presented the Service Composition algorithms proposed and evaluated in this Thesis. These algorithms are based on two notable automated planning algorithms: Graphplan and Fast-Forward (FF). AI planning, as pointed out in Chapter 3, is a recurrent approach for implementing Automated Service Composition. In fact, the Graphplan algorithm was already employed in previous works, e.g., [97] and [95]. Nonetheless, none of them, to our knowledge, addressed the problems we tried to solve using Graphplan, namely, enforcing the use of the composition inputs specified by the developer and allowing incomplete input specifications. The first feature aims at improving the adherence of the solution found to the specification and find solutions closer to the ones a human developer would build. It can be seen as a special case of Planning with Preferences [6] – where the developer can define additional constraints the resulting plan should meet, e.g., optional goals (outputs) or specific intermediate states the plan should traverse preferably. One kind of preference is in the trajectory of the plan, i.e., the states traversed from the initial state (inputs) to the goal (outputs) by executing the plan (composition). Our algorithms implement a hard constraint on the trajectory, forcing the solution to pass through the initial state as specified by the developer (a soft constraint, on the other hand, would just penalize with higher cost solutions that violate it). With the "missing inputs" feature, on the other hand, we try to be more permissive with the developer, who will not need to know all inputs in anticipation. Given its ability to handle an uncertain initial state (a missing input implies that the initial state may not be that specified by the developer), our algorithms can be seen as conformant planning algorithms [25], although strictly adapted to service composition. A conformant planner tries to remove uncertainty by forcing the system into a known state – in our case, by adding a new input parameter – and then finding a solution to the problem. The performance of the algorithms presented in this chapter, both in terms of computational time and quality of the solutions, is discussed in the next chapter.

Chapter 6 Performance Evaluation

This chapter presents the experiments carried out to evaluate the performance of the algorithms described in the previous chapter. In this evaluation, we investigate how state-of-the-art planners perform when applied to the service composition problem with respect to the quality of the solutions as well as the time required to obtain them. The planners' performance is compared to the proposed algorithms' and the effects of the algorithm features are discussed. First, an overall description of the experiments including metrics and evaluated planners is show in Section 6.1; Section 6.2 brings the evaluation results and their explanations; Section 6.3 wraps up this chapter with a review of the results and overall discussion.

6.1. Evaluation Preliminaries

In our approach for Automated Service Composition, we are concerned with the practical aspects that hinder the application of automated solutions in real life. We have discussed the lack of semantic descriptions as one of this problems, which we addressed using a reverse engineering approach. We see automated composition as an auxiliary tool for the service developer – not a replacement. Therefore, it is expected that the developer will have to deal with the output of such automated tools. The usefulness of these tools depends on how much effort can be saved by applying them. In this evaluation, instead of directly assessing the effort gain or loss, we approximate it by measuring how close to handwritten compositions are the automated solutions found by the algorithms. Since we have the original compositions from the myExperiment repository at hand, we are able to compare the output of the algorithms with the original compositions for the same initial specification. The more similar are the solutions to the original compositions, the less effort the developer would have to spend adapting and fixing them. We identified the factors that influence the quality of the solutions and improved our algorithms based on them. In order to have a baseline for comparison, we selected various classical and state-of-the-art planners from the AI planning literature, from which one was used in the experiments as our baseline, as explained as follows.

6.1.1. Automated Planners

There are several planning tools described in the literature, some of them can be downloaded freely. We collected a representative set of planning tools in order to make up our evaluation baseline. Among the planners available, we tested the following tools: the Fast-Downward planner [32], the original Fast-Forward [34], Satplan [42], POPF2 [14], SGPlan [38], MIPS-XXL [19], and the Graphplan implementation present with the PDDL4J library [69]. Other classical tools, often used as a reference in planning works – the original Graphplan [11], Blackbox [41] and IPP [48] – have not been updated in the last years and therefore were not included in the evaluation. All these tools accept as input the de facto language for automated planning, PDDL. The selected tools were installed and tested against PDDL descriptions of our service composition scenarios with varying sizes. From all these tools, the only one capable of running with scenarios of significant size (more than 100 workflows) was the Fast Downward planner; all other tools presented some problem that prevented their use in representative scenarios (most frequently, memory overflows or violations), although were capable of running with the smaller test samples which accompanied them.

The Fast Downward (FD) planner was selected initially because of its performance in the last International Planning Competition [40] (other selected planners that participated in previous IPC editions were POPF2, SGPLAN and MIPS-XLL). The FD planner, which has as contributor the author of the FF planner, is in fact a flexible platform that supports various heuristics and search strategies. In the IPC 2011 competition it participated under several names, for each heuristic (or set of heuristics) used, and obtained the first places in the categories it participated in. As the FF planner, the Fast Downward is a forward planner, i.e., it searches the state space from the initial state until a goal state is found. Instead of using the planning graph as a heuristic as in FF, the FD planner adopts another structure called "causal graph" as the base structure for computing heuristic values. The causal graph is defined by causal dependency relationships between objects of the planning domain: two objects have a causal dependency if the state of the first object is changed by an operation whose precondition depends on the state of the second.

We used three FD heuristics (or variations) from the IPC2001 in our evaluation: FDSS1 [33], SelMax [18] and LAMA2011 [74]. The FDSS1 (Fast Downward Stone Soup 1) is in fact a meta-heuristic that combines different heuristics and search strategies into a sequential planning portfolio, and is based on the premise that "no single common search strategy or heuristic dominates all classical planning problems" [33]. The FDSS combines A* search (and variants) with existing FD heuristics for assigning weights to states, including strategies employed by other participating planners, such as the idea of "landmarks" used by the LAMA2011 heuristic. Landmarks are (partially) ordered sub-goals that must hold at some point for every plan found, i.e., for some specific planning problem there may be implicit "milestones" for the task to be accomplished, irrespective of the solution the planner may eventually find. The other heuristic used, SelMax, is similar to FDSS1 in the sense that it is a meta-heuristic: it tries to guess which heuristic is more adequate for evaluating each space, while trying to avoid computing several heuristics. It uses a learning process to create a rule for deciding which heuristic to apply at each search step. It is important to note that all these heuristics assume that "costs" are associated to actions (services), and that the total cost of the solution must be minimized. However, in our scenarios no cost information exists, therefore the task of the heuristics comes down to finding a plan that achieves the goal specified (i.e., provides the output parameters). The use of costs for the services – e.g., based on QoS measurements – is a subject for future work.

6.1.2. PDDL Generation

PDDL descriptions of our service composition scenarios had to be generated in order to use the planning tools mentioned in the previous section. PDDL is a standard language widely supported by planning tools that provides several features, depending on the degree of detail of the planning domain being specified. The support of these features is, however, very uneven among planners, and a more direct mapping from the concepts in the Service Composition domain to structures in the PDDL language – e.g., using PDDL types to model parameter types – could not be used (`:typing`, in PDDL terminology, is not supported by all selected planners). A "common denominator" specification format had to be found so that more tools could be tested. We avoided using different specification formats for each planner to prevent the results from being biased by the formats. After several preliminary tests, we found a pure STRIPS (`:strips`) PDDL specification – with predicates representing parameter types and their compatibility relationships – that was supported by the majority of the planners tested.

The mapping to PDDL resembles the example given in Section 5.1 of the Proposed Algorithms chapter, but with some notable differences. In that example, a banking service $SetBalance$ accepts two parameters as input – a $AccountId$ and either a $Real$ or $Integer$

value as the new balance – and returns a variable of type $Boolean$ indicating if the operation was successfully performed. The input parameters are mapped to preconditions of the *SetBalance* action and the returned value becomes its effect. The predicate $available(x)$ is used to indicate that a variable of type x is available. In the PDDL generation, the $available$(x) predicate – which becomes `(available ?x)` in PDDL notation – is used with parameters of the PDDL action associated to the services. A predicate is created for each existing type in the database – for example, `(t_real x)` to indicate that a variable x is $Real$ – and these predicates are used in the action precondition to ensure correct type matching. The generated PDDL code for the $SetBalance$ service is shown in Figure 6-1. Due to limitations in PDDL support of the planners, disjunctive preconditions could not be used and an alternative approach was employed. In the example in Figure 6-1, we have a new type $Balance$ replacing the "$Real$ or $Integer$" construction of the original example.

```
(:action SetBalance
       :parameters (?x ?y)
       :precondition (and  (available ?x) (t_accountId ?x)
                           (available ?y) (t_balance ?y))
       :effect (and (available Boolean) (t_boolean Boolean))
)
```

Figure 6-1. PDDL code for the SetBalance service.

The type compatibility information is generated in the effect of the actions, allowing output variables to be used as input to various services by assuming different types. For example, consider the $GetBalance$ service, which receives an $AccountId$ as input and returns the balance as a $Real$ number. Let us assume that, according to the reverse engineered model, the $Real$ type is compatible with $Balance$. The effect of the action $GetBalance$ must state that the output variable $balance$ is available and that it has type $Real$ and also $Balance$ (`t_balance` predicate). This way, a call to the service $GetBalance$ can be connected by the planner to a call to $SetBalance$ via the $balance$ variable. The code for $GetBalance$ is show in Figure 6-2.

```
(:action GetBalance
       :parameters (?x)
       :precondition (and (t_accountId ?x) (available ?x) )
       :effect (and (available Balance)
                    (t_real Balance)
                    (t_balance Balance))
)
```

Figure 6-2. PDDL code for the GetBalance service.

The set of actions generated from the services, the type predicates and variables used as outputs of the actions comprise the PDDL domain specification, a single text file submitted to the planner with the problem specification, one per composition problem to be solved. A PDDL problem specification states the initial and goal states for plan computation. In our translation to PDDL, initial states take a form similar to action effects: the composition inputs become the variables initially available declared with all compatible types. On the other hand, the goal state resembles an action precondition, with the exception that action variables such as "?x" are not allowed in the goal state. The PDDL specification for a composition problem where an $AccountId$ is given as input and a $Boolean$ expected as output is shown in Figure 6-3.

```
(define (problem prob1)
       (:domain banking)
       (:objects AccountId Balance Real Integer Boolean )
       (:init (available AccountId) (t_accountId AccountId))
       (:goal (and (available Boolean) (t_boolean Boolean)))
)
```

Figure 6-3. Problem specification in PDDL.

In case of success, the planner outputs a plan (or set of plans) that fulfills the problem specification. The plan is a list of actions invocations with the parameters used as inputs to the action; no information on the output is provided. In our experiments, we translated the PDDL plans generated by the planners to our internal representation where quality metrics were computed. Figure 6-4 shows the plan rendered by the baseline planner for the domain and problem specifications described in this section.

```
(GETBALANCE ACCOUNTID)
(SETBALANCE ACCOUNTID BALANCE)
```

Figure 6-4. PDDL plan generated for the example.

6.1.3. Evaluation Methodology

The main evaluation scenario (Main scenario) consists of rebuilding each original composition from the myExperiment repository by submitting its interface description, i.e., input and output parameters, to the planners and algorithms and measuring the quality of the solution found. Given the limitations imposed by the baseline planner (FD), we had to reduce the set of compositions from 425 to 300 elements (no planner was able to run with 400 or more compositions within our resources). We used the 300 compositions with lowest Id number from the initial set of 425 compositions to create the repository of services, types and parameters, as described in Chapter 4. However, not all compositions from the 300-set are present in the results. Some cases were excluded to create a more realistic scenario to the algorithms and prevent distortions in the results: compositions with no output parameters (not suitable for the planners or algorithms); compositions containing one single service; "isolated" compositions, whose all services are not used by any other composition in the repository (the term "isolated" refers to the small disconnect sub-graphs visible in Figure 4-7).

The algorithms were also evaluated against a set of 100 randomly generated composition specifications (Random dataset). Each specification defines a composition interface with 0 to 3 input and 1 to 3 output parameters randomly selected from the set of parameters in the repository. The objective of this scenario is to test the algorithms with problems that the service repository is not "familiar" with, i.e., cases that were not used to generate the semantic information in the repository in the first place.

The experiments were conducted in a pool of four machines distributed between GPRT/UFPE and UFABC. Each machine has a quad core Intel Xeon processor and 12 GB RAM memory. Each individual experiment was allowed to use up to 8GB of memory and run for 30 minutes (time limit used in IPC competitions). The algorithms presented in Chapter 5 were implemented in Java and executed with the same time and memory constraints. The basic Graphplan algorithm from the literature adapted for service composition is identified in the evaluation as GP. The GP variant proposed in this Thesis that supports enforced and missing inputs is identified by $GP(\Delta_{max}, \Gamma_{max})$ (in fact, the original GP is equivalent to $GP(\Delta_{max} = \infty, \Gamma_{max} = 0)$, i.e., no limit to the number of unused inputs, $\Delta_{max} = \infty$, and no missing parameters allowed, $\Gamma_{max} = 0$). We use the shortcut $GP_{enf} = GP(\Delta_{max} = 0, \Gamma_{max} = 0)$ to refer to the use of enforced inputs, and $GP_{miss} = GP(\Delta_{max} = 0, \Gamma_{max} = 1)$ to indicate the support for missing inputs (in our experiments, only one missing parameter was allowed and the remaining inputs enforced). Our Fast Forward (FF) variations follow the same notation. The superscript x^{pref}indicates the use of Preferred Services with the algorithms, for example, GP_{enf}^{pref} denotes the Graphplan algorithm with enforced inputs and preferred services.

6.1.4. Quality Metrics

The quality of a composition can be a very subjective matter. In our evaluation, the quality of a composition generated automatically is a measure of how similar to a handwritten composition it is for the same initial specification. We compared the solutions provided by the baseline planner and our algorithms to the corresponding composition in the repository to determine their similarity degree. The similarity between compositions itself can be measured with different levels of detail. Instead of a very precise similarity measure – e.g., considering individual connections between services and the order of services inside of the compositions –, we opted for a high-level similarity metric that uses only the list of services present in both compositions, irrespective of the order or number of occurrences of these services within the compositions.

One reason for using only the list of services is that finding the correct services for a given composition represents a fundamental part of the process of creating a new service composition manually. If an automated composition process could generate compositions with the correct services, even if not connected in the way the developer would expect, this could save him/her a considerable amount of time. Another reason is to ensure a fair comparison between the planners and our algorithms. While a PDDL plan – given as solution to a composition problem – contains an ordered list of services that fulfill the composition specification, it lacks the information on how the services are connected exactly (and whether they can be executed in parallel or not, for example). Our translation algorithm guesses these connections (from outputs to inputs) in order to generate a valid composition in accordance with the plan. Therefore, taking into consideration these "guessed" connections in the similarity metric could distort the results.

The similarity metric we adopted considers the percentage of services correctly found by the composition process with respect to the total services on both compositions, automatic and manual. Formally, the similarity between two compositions C_1 and C_2 is given by formula (6-1):

$$similarity(C_1, C_2) = \frac{|services_{C_1} \cap services_{C_2}|}{|services_{C_1} \cup services_{C_2}|} \tag{6-1}$$

This metric ranges between zero and one, where $similarity(C_{orig}, C_{manual}) = 0$ means that the automatic (C_{auto}) and manual (C_{manual}) compositions do not share any services, while $similarity(C_{orig}, C_{manual}) = 1$ means that both compositions contain the exact set of services. A closely related metric we employed is the "hit rate", equation (6-2), which reflects the number of correct services ("hits") found.

$$hit(C_{auto}, C_{manual}) = \frac{|services_{C_{auto}} \cap services_{C_{manual}}|}{|services_{C_{manual}}|} \tag{6-2}$$

Conversely, the "error rate" measures the number of "false positives", i.e., services that the composition algorithms found useful but the developer will likely remove from the final composition (eq. (6-3)).

$$err(C_{auto}, C_{manual}) = \frac{\left|services_{C_{auto}} - services_{C_{manual}}\right|}{\left|services_{C_{auto}}\right|} \quad \textbf{(6-3)}$$

Other simpler metrics were collected during the experiments – average number of services per composition, depth of the composition, etc. – as well as performance indicators: time to compute each solution, number of successful and failed problem instances, and cases that extrapolated the time limit.

6.2. EVALUATION RESULTS

6.2.1. Basic Evaluation

The first scenario we evaluated aims at measuring the performance of standard planning tools with respect to both quality and speed. We compared the FD variants – $FDSS1$, $SelMax$ and $Lama2011$ – to our standard Graphplan implementation (referred to as GP), using the 300-compositions dataset. Table 6-1 summarizes the results for these first experiments, showing the number of cases where each planner succeeded, failed or was terminated due to timeout, along with the average time per experiment (plus/minus the standard deviation).

Table 6-1. Evaluation summary for the Main scenario.

	$FDSS1$	$SelMax$	$Lama2011$	GP
Success	199	122	194	212
Failed	3	1	4	0
Timeout	10	89	14	0
Time (avg±sd)	**519±282s**	**383±59s**	**516±243s**	30±65ms

In this scenario, we are trying to rebuild the compositions using the information that was extracted from them in the first place; therefore, we should expect the tools being able to find solutions to all experiments. The first notable observation is that only the GP algorithm was capable of that; the FD-based heuristics failed for some compositions due to timeout or were not able to find a solution at all (before timeout). Among the FD heuristics, the $SelMax$ had the worst success rate, failing in more than 40% of the cases (including timeout cases); its companion heuristcs had failure rates below 10%.

Another important observation is the time required by the FD planners to compute the solutions. Note that, in Table 6-1, while the times for FD planners are shown in seconds (**highlighted** for easy differentiation), the GP algorithm computed solutions in tens of milliseconds in average. This significant difference, however, needs to be analyzed carefully. General-purpose planners, such as the FD, provide several features, for various planning domains, which are not supported by our GP version. In addition, they are optimized for solution quality (minimum cost) since the IPC competition uses quality as its most important metric. Other details contribute to the excessive time, for example, the parsing of PDDL descriptions, the generation of large intermediate files, and the use of interpreted languages such as Python in some stages. Nonetheless, this result suggests that a specialized implementation can be several orders of magnitude faster than a general-purpose planner can, and therefore preferred for use in production scenarios.

Completing this evaluation, Table 6-2 shows the planners performance in the Random scenario ($FDSS1$ and GP only), which helps estimating the time the developer would wait, in average, for failed cases. In general, time figures are similar to the Main scenario for the $FDSS1$ heuristic, while the GP algorithm needed 50% more time in average, yet much faster than the FD planner.

Table 6-2. Evaluation summary for the Random scenario.

	$FDSS1$	GP
Success	17	25
Failed	75	75
Timeout	8	0
Time (avg±sd)	**465±142s**	45±56ms

The quality of the solutions, measured by the similarity metrics described in Section 6.1.4, is presented in Table 6-3. Overall, the quality of the solutions generated by the FD heuristics and the GP algorithm are around 75%, with the exception of the $SelMax$ heuristic, which needs further explanation for its 0.88 quality. It must be noted that this heuristic has the worst performance in terms of success rate, being able to solve only 60% of the cases tested. In order to provide a fair comparison between planners, Table 6-4 shows the similarity of the solutions for the cases where both the FD heuristic and the GP had succeeded, for the three heuristics evaluated. In the case of $SelMax$ versus GP, both offered the same similarity of 88% for the cases both found a solution. In general, the GP performed slightly worse than the FD heuristics given its tendency to add more services than necessary, as shown by the average number of services of its solutions: 4.74. This happens since the Graphplan algorithm tries to produce plans with fewer levels (each level containing potentially many services), irrespective of how many services the plan will end up with.

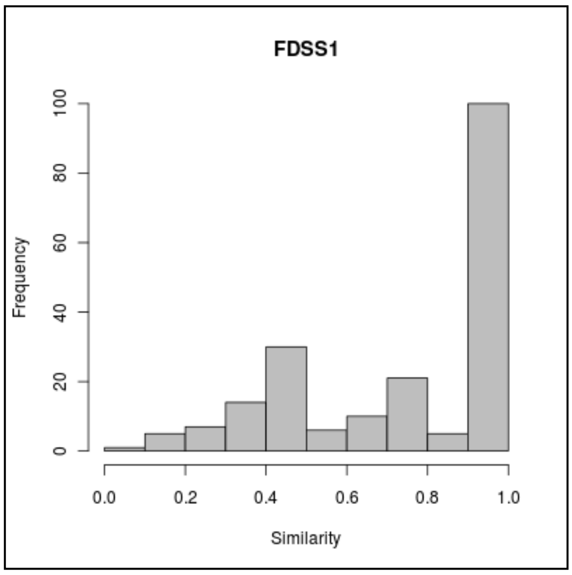
Table 6-3. Composition metrics for the Main scenario.

	$FDSS1$	$SelMax$	$Lama2011$	GP
Similarity	0.77	0.88	0.75	0.73
Hit rate	0.79	0.90	0.77	0.78
Error rate	0.08	0.04	0.10	0.14
Services (avg)	4.14	4.18	4.07	4.74
Depth (avg)	3.24	3.27	3.25	3.33

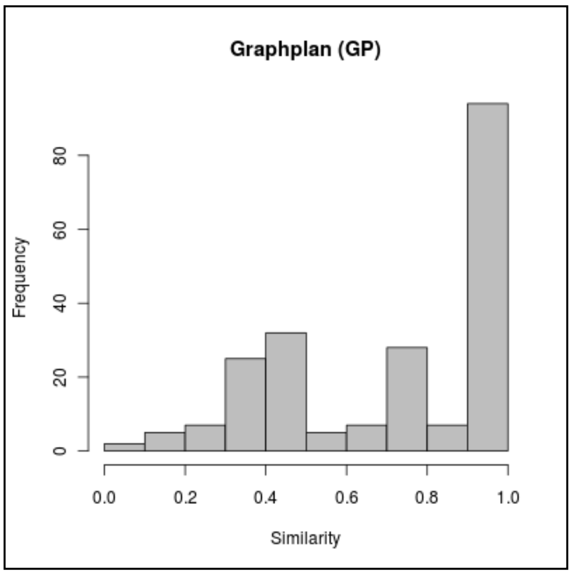
Table 6-4. Similarity: FD heuristics vs. GP for the cases where both succeeded.

	$-$	GP
$FDSS1$	0.77	0.74
$SelMax$	0.88	0.88
$Lama2011$	0.75	0.74

Overall, the quality of the solutions can be considered acceptable – around or superior to 70% – for the planners with good success rate. Figure 6-5 shows the histogram for the similarity metric, where it is shown that the $FDSS1$ algorithm concetrates 50% of its solutions in the 1.0 extreme of the similarity spectrum; the GP algorithm presents a similar behavior, concentrating slightly more solutions in the middle of the range than $FDSS1$.



(a)



(b)

Figure 6-5. Similarity histogram for FDSS1 (a) and GP (b).

It can be noted that for a percentage of the cases the average quality degrades to as low as 45%. 35 of these cases (around 16% of the total) have as a common characteristic the fact that not all input parameters provided by the developer in the composition specification were used to build the final solution. In fact, the difference between the number of inputs provided in the specification and the number of inputs actually used has a linear correlation of -0.46 with the quality of the solutions, suggesting that decreasing this difference would provide more accurate compositions. As mentioned before, general-purpose planners focus in finding the shortest plan (in number of services or time steps); using all the inputs provided is not a priority. To illustrate this fact we present in Figure 6-6 an example of workflow from the myExperiment repository and, in Figure 6-7, the respective solution found by both the $FDSS1$ and GP planners.

The workflow in Figure 6-6 uses three data Web Services (marked in the figure with "wsdl:" prefix) and internal Taverna activities for processing their input and output parameters; for the purpose of this Thesis, both Web Services and internal activities are handled as "services". The processing activities in this case – $XMLInputSplitter$ and $XMLOutputSplitter$ – concatenate and separate individual parameters into and from XML code, respectively. Alghouth the Web Services in this example require XML-encoded parameters, this is not required for the Web Services in the myExperiment repository. The solution for this case, shown in Figure 6-7, does not use all the original input parameters – $NumRunYear$ is leaft unused. As a result, it lacks one of the Web Services of the original solutions – $wsdl{:}RunDynamicSimple$ – and the solution quality is 0.67.

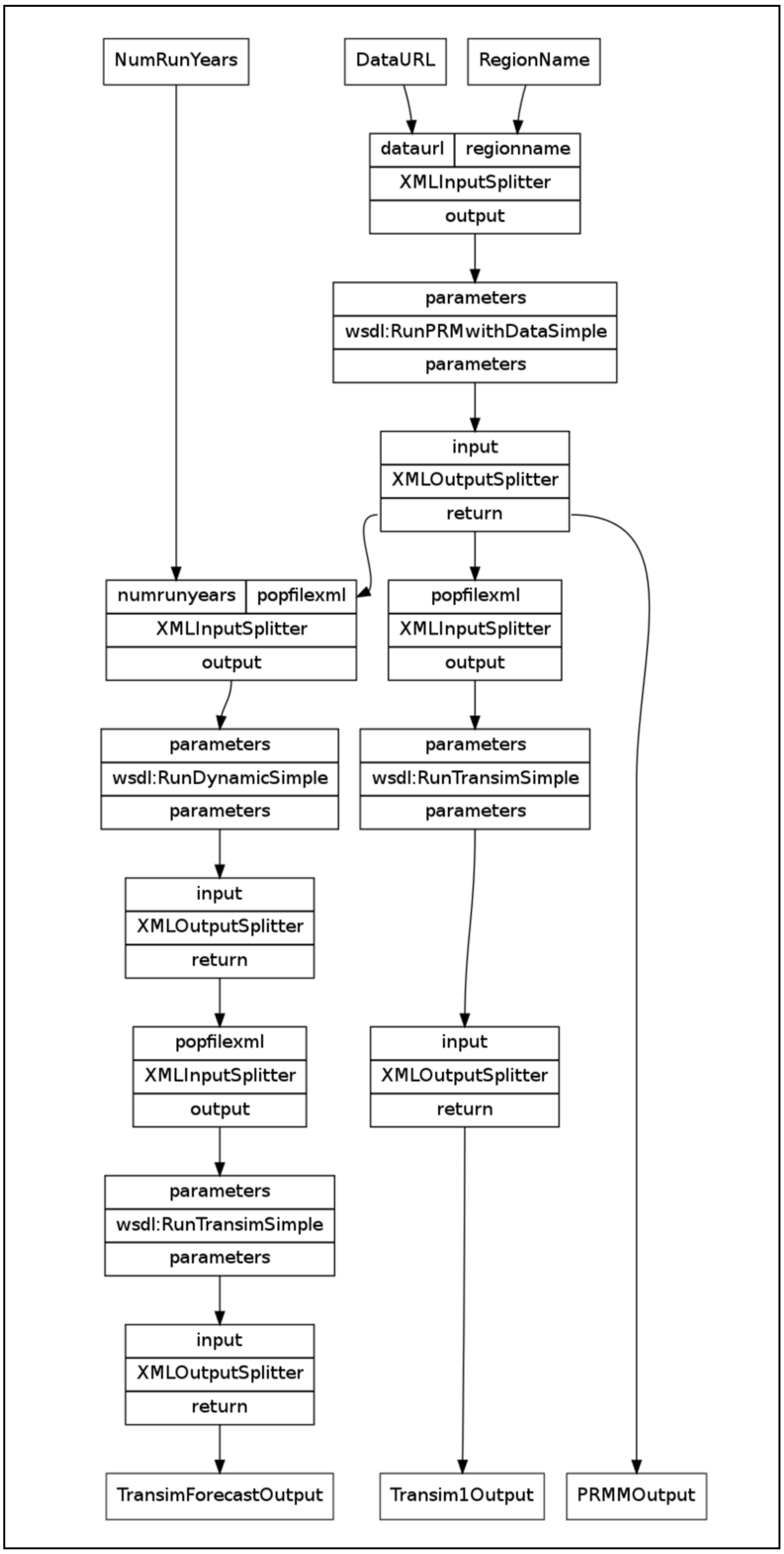


Figure 6-6. Example of workflow from the myExperiment repository.

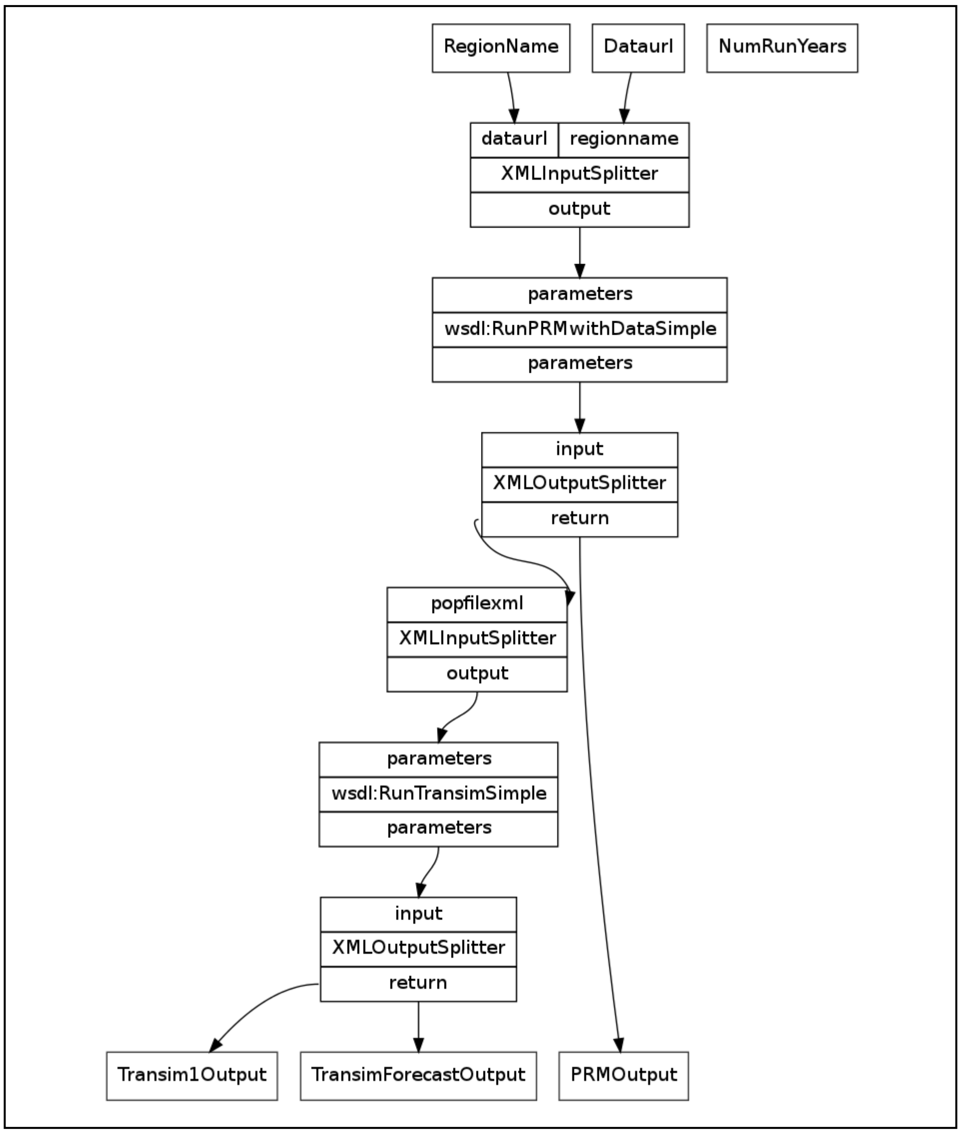


Figure 6-7. Resulting workflow using FDSS1 or GP for the previous example.

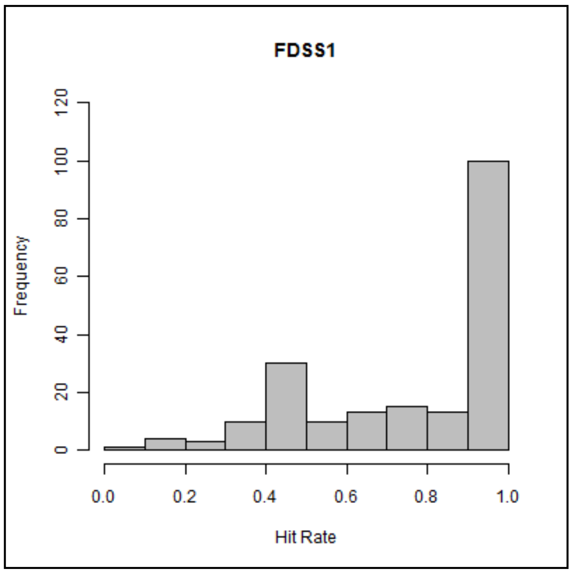
6.2.2. Enforcing Inputs to Improve Quality

We addressed this problem with extensions that enforce the use of the inputs provided by the developer, as described in the Chapter 5. We tested these algorithms in the Main scenario, comparing it with the $FDSS1$ heuristic. In Table 6-5, we present the results for the enforced GP and our FF variants, with and without the use of Preferred Services; the results for $FDSS1$ and basic GP are shown for easy reference. With respect to the FF algorithm, we opted to show only the results using the modified A* search strategy, which performed slightly better than the standard A* in our tests; using the Enforced Hill Climbing (EHC) search routine from the original FF algorithm led to a high failure rate. (The original FF implementation already accounted for that and used a secondary search strategy to deal with failed cases [11].)

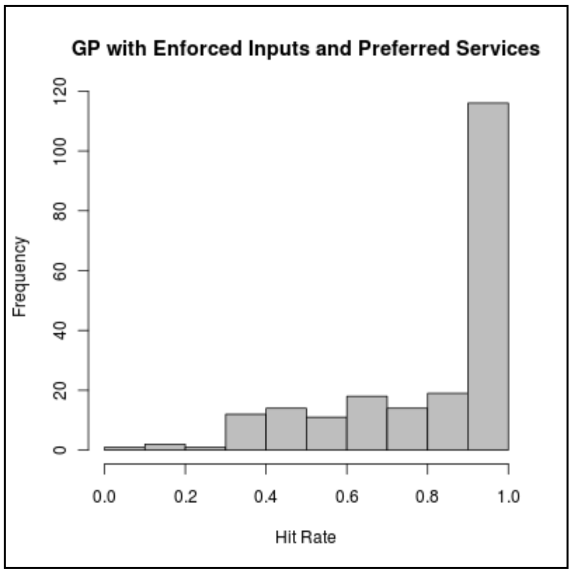
Table 6-5. Composition metrics enforcing the use of input parameters.

	$FDSS1$	GP	GP_{enf}	GP_{enf}^{pref}	FF_{enf}	FF_{enf}^{pref}
Similarity	0.77	0.73	0.78	0.77	0.79	0.78
Hit rate	0.79	0.78	0.83	0.83	0.84	0.83
Error rate	0.08	0.14	0.12	0.12	0.11	0.12
Services (avg)	4.14	4.74	4.92	5.01	4.96	4.94
Depth (avg)	3.24	3.33	3.44	3.49	3.46	3.42

Examining only at the quality obtained by the algorithms, the improvement seems unimpressive. The modified algorithms matched the quality provided by the FD planner, being also able to improve the hit rate – i.e., number of correct services found – from 0.79 to 0.84, as can be seen in the histograms in Figure 6-8.



(a)



(b)

Figure 6-8. Hit rate histograms for $FDSS1$ and GP_{enf}^{pref} (b).

Table 6-6 gives a more complete panorama of the algorithms´ performance. The GP_{enf} algorithms showed capable of solving most of the problems (97%) with slightly better quality – especially hit rate – than the $FDSS1$ heuristic; the use of preferred services allowed 2 cases that timed out with GP_{enf} to be completed successfully, but also increased considerably the average time to compute the solutions due to the initial step of computing the set of preferred services. Both of our FF variants, FF_{enf} and FF_{enf}^{pref} had success rates compatible with $FDSS1$ but with little gain in solution quality.

Table 6-6. Success and time metrics when enforcing the use of input parameters.

	$FDSS1$	GP	GP_{enf}	GP_{enf}^{pref}	FF_{enf}	FF_{enf}^{pref}
Success	199	212	206	208	195	199
Failed	3	0	3	3	15	12
Timeout	10	0	3	1	2	1
Time (avg±sd)	**519±282s**	29±65ms	31±63ms	**2±19s**	104±307ms	134±432ms

The real value of the proposed algorithms surfaces when we look only at the cases where they made a difference: instances where traditional planning algorithms did not use all the information provided by the developer in the solution. The average quality for these cases rose from 0.45, using the $FDSS1$ heuristic, to 0.71 using the enforced GP algorithm with preferred services (GP_{enf}^{pref}), as can be seen in Table 6-7. The hit rate also increased considerably: while the FD planner found around half the services the developer wanted, our algorithm was able to find as many as 80% of them.

Table 6-7. Composition quality when enforcing the use of input parameters; only cases affected by the algorithms.

	$FDSS1$	GP_{enf}^{pref}
Similarity	0.45	0.71
Hit rate	0.54	0.82
Error rate	0.31	0.18
Services (avg)	3.77	5.20
Depth (avg)	3.03	3.91

Resuming the example of the previous section, Figure 6-9 shows the resulting composition when enforcing the use of all input parameters. The solution contains all the services of the original workflow (Figure 6-6), and only them, what gives it a quality measure of 1.0. However, it must be noted that the order and number of service invocations differs from the original, what is not reflected by our similarity (quality) metric. Although a solution still requires the intervention of the developer in order to take its final form, the task of finding the services was performed correctly by the composition algorithm.

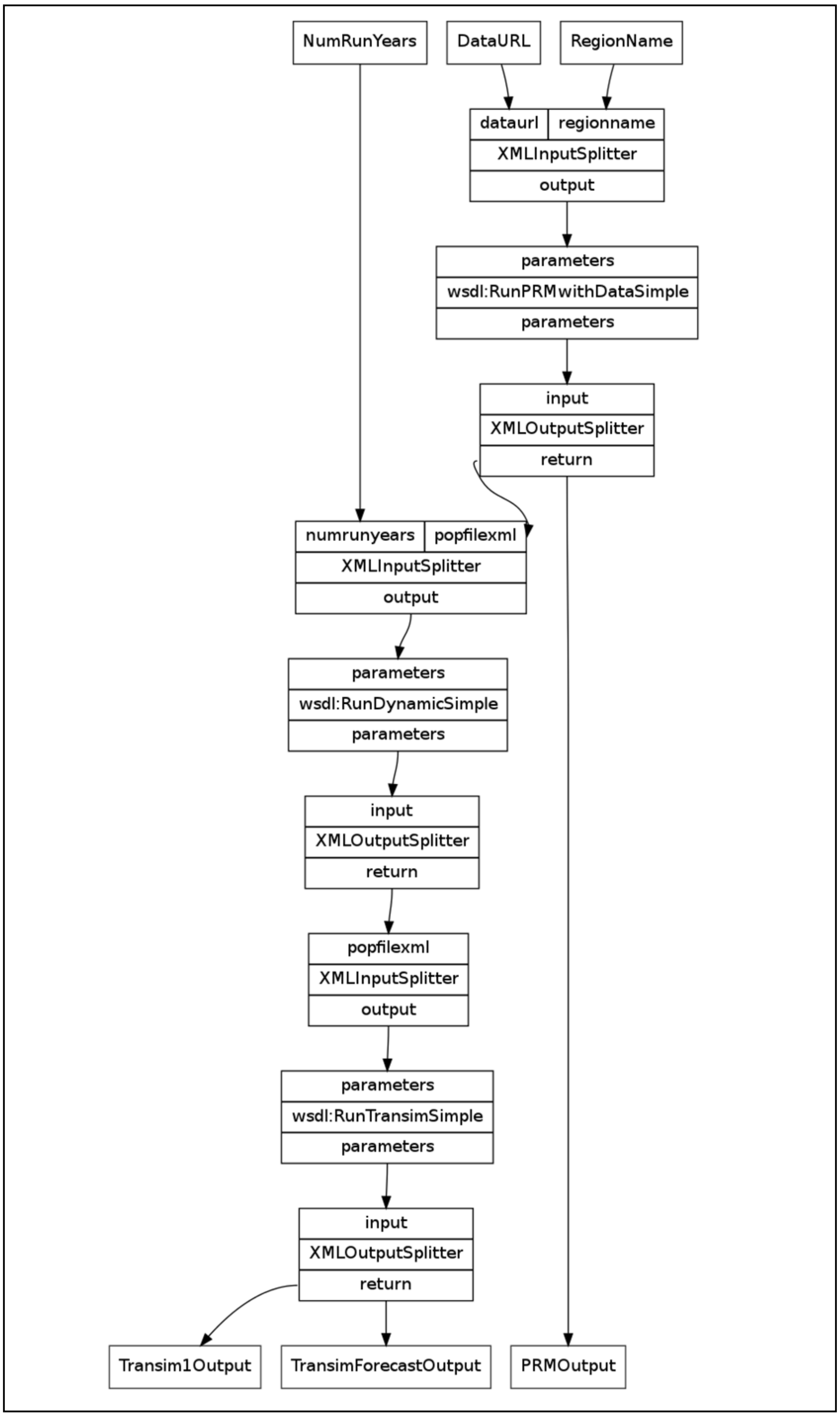


Figure 6-9. Resulting workflow enforcing the use of all inputs.

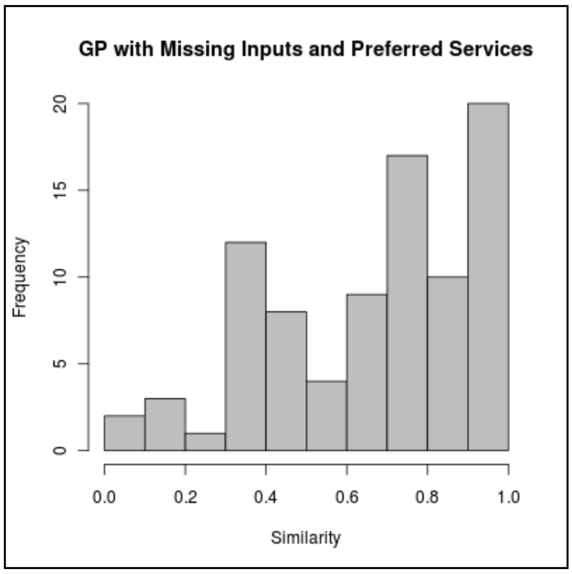
6.2.3. Missing Inputs

Another feature explored in the proposed algorithms is the tolerance of incomplete specifications, which in our case means that some input parameters needed by the composition may be missing in the description provided by the developer. Both our adapted GP and FF algorithms have been changed to handle such cases. In order to test this feature, we modified the Main scenario so that the specifications used to (re)build the compositions contained one input parameter less than originally (cases with no inputs where not included in this experiment). The $FDSS1$ planner and basic GP algorithm were applied to this scenario for generating a reference solution. Table 6-8 compiles the results of this battery of experiments;

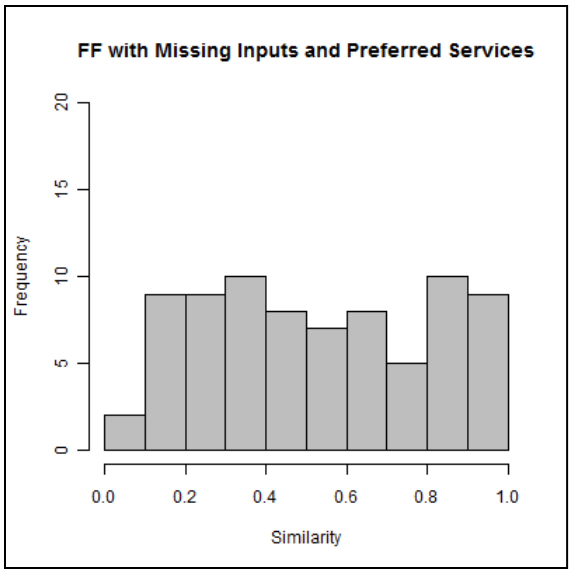
Table 6-8. Composition metrics for the scenario with one missing input parameter.

	$FDSS1$	GP	GP_{miss}	GP_{miss}^{pref}	FF_{miss}	FF_{miss}^{pref}
Similarity	0.58	0.55	0.69	0.68	0.54	0.55
Hit rate	0.65	0.64	0.76	0.76	0.58	0.58
Error rate	0.20	0.25	0.15	0.15	0.10	0.11
Services (avg)	4.53	5.32	5.53	5.53	3.89	3.88
Depth (avg)	3.55	3.71	3.63	3.63	2.48	2.48

While the quality of the solutions suffers with the removal of one input parameter, the drop was less pronounced than expected. The average similarity fell from the initial 0.73 with basic GP (in the Main scenario) to values bordering 0.7 using both GP_{miss} and GP_{miss}^{pref}. The FF variants performed slightly worse than $FDSS1$ in this scenario, spreading the solutions all over the similarity spectrum as shown in Figure 6-10(b).



(a)



(b)

Figure 6-10. Histogram of similarity for GP_{miss}^{pref}(a) and FF_{miss}^{pref}(b).

However, the goal of the proposed algorithms is to provide approximate solutions to the developer in cases where traditional approaches would fail straightway, as seen by the results of the FD and GP planners in Table 6-9. The success rate of these planners was limited to 66% (63 successful cases out of 95 for GP). Meanwhile, our FF-based algorithms had up to 81% success rate and both GP_{miss} and GP_{miss}^{pref} solved 90% of the cases.

Table 6-9. Success and time metrics with one missing input.

	$FDSS1$	GP	GP_{miss}	GP_{miss}^{pref}	FF_{miss}	FF_{miss}^{pref}
Success	58	63	86	86	73	77
Failed	33	32	7	7	22	18
Timeout	4	0	2	2	0	0
Time (avg±sd)	**435±74s**	42±80ms	58±95ms	163±398ms	**0.4±2.3s**	199±361ms

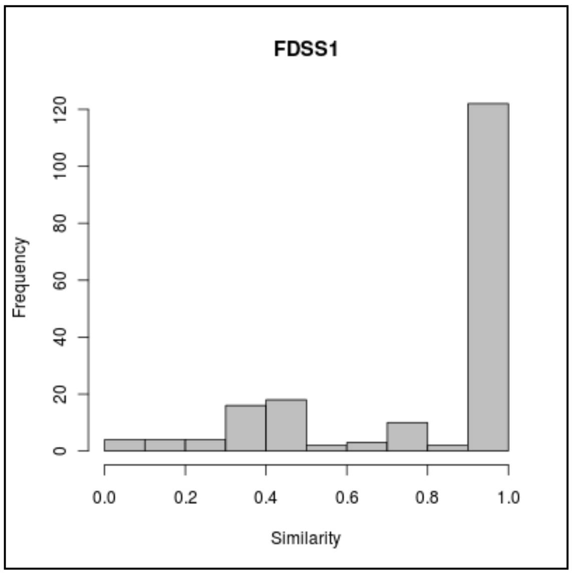
6.2.4. Service Precedence

To finalize our evaluation, we experimented with the information gathered from the compositions in the reverse engineering phase. In particular, we are interested in measuring the influence of using the service precedence information in the composition process. The service precedence relation, as described in Section 4.2.2, can be used to determine if a service X is a prerequisite for service Y, even if they are not directly connected. Intuitively, if we observe that service X precedes service Y in all compositions were service Y is used then one can assume that X is needed in order to invoke Y, i.e., X is an implicit precondition of Y. An example of this kind of relationship is the case of a service that closes a database connection. It must clearly be preceded by an invocation to the service that opens the respective connection at some point of the composition. For this evaluation, we made the precedence relation an explicit precondition of the services in our repository. No modifications were made to the algorithms since they are able to handle preconditions natively. We evaluated the Main scenario with precedence information using the $FDSS1$ planner and the basic GP algorithm, the results shown in Table 6-10 (along with the results without precedence).

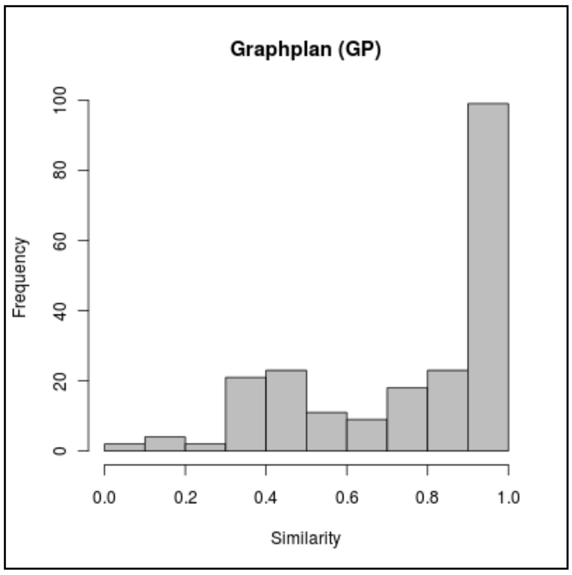
Table 6-10. Composition metrics with and without the use of service precedence.

	Without service precedence		With service precedence	
	$FDSS1$	GP	$FDSS1$	GP
Similarity	0.77	0.73	0.81	0.77
Hit rate	0.79	0.78	0.83	0.86
Error rate	0.08	0.14	0.09	0.14
Services (avg)	4.14	4.74	4.13	5.76
Depth (avg)	3.24	3.33	3.35	3.53

The use of the additional information had an impact on the quality of the solutions, breaking the 80% barrier with $FDSS1$. It also generated more solutions close to 1.0 than the GP algorithm, as seen in Figure 6-11. Graphplan outperformed $FDSS1$ with respect to the hit rate metric (0.86 against 0.83) but suffered from the large number of services in its solutions, which reduced its overall similarity.



(a)



(b)

Figure 6-11. Histogram of similarity for FDSS1 (a) and GP (b) with Service Precedence.

This gain in quality comes with a price: it takes 150% longer to compute the solutions with $FDSS1$, and more than 10 times with Graphplan (Table 6-11). $FDSS1$ also fails more frequently due to timeout once the average time of 22 minutes approaches the hard limit set for our experiments: 30 min.

Table 6-11. Success and time metrics with and without service precedence.

	Without service precedence		With service precedence	
	$FDSS1$	GP	$FDSS1$	GP
Success	199	212	185	212
Failed	3	0	1	0
Timeout	10	0	26	0
Time (avg±sd)	**519±282s**	29±65ms	**1332±62s**	388±1006ms

In our experiments, we used service precedence as a "hard" precondition: a service is only added to a solution if all services it requires appear before in the solution its invocation. In the future, we plan to use the precedence information in a "softer" way, deferring instead of pruning away the services that have not all of their required services fulfilled. We find this soft version more interesting in general, since the precedence information gathered via reverse engineering can be too tied to the specific compositions used in the process, and therefore too restrictive in a general sense.

6.3. Summary and Discussion

This chapter presented the performance evaluation intended at measuring the quality of the compositions provided by a standard planner and by our proposed algorithms. We compared the solutions found by those tools with the original, handwritten compositions of the myExperiment repository in order to estimate how helpful these tools would be to composition developers. The results showed that the overall quality of compositions obtained by both standard planning tools and our algorithms is acceptable (around or above 75%), but there is still room for improvement for specific cases. We identified that the cases where not all input information was used in the solutions had lower average quality, and that by applying an algorithm designed to address this problem the quality was increased from 45% to 71%. The tolerance of our algorithms for missing inputs was also evaluated. We were able to raise the success rate of the automated service composition from 66% to 90%, compared to classical planning, while keeping the quality of the solutions bordering the 70% level. The results also showed that a tailored implementation can perform orders of magnitude faster than a more general one while providing similar quality, although this finding comes with some reservations. Another interesting result was that using more information associated to the services (i.e., service precedence) can improve the solutions at the expense of significantly increasing the time required to compute them.

The decision of what composition tool to use – a generic planner or our tailored algorithms – vary according to the requirements of the composition developer. If the developer has little or no constraint on the time required to compute the solutions, using a generic planner such as the Fast Downward along with precedence information (as in Section 6.2.4) is a good option since is generates compositions with best quality in average (but takes several minutes doing so). However, if response time is an issue, as in an iterative development approach, the results point that our algorithms, in especial the Graphplan with Enforced Inputs and Preferred Services (GP_{enf}^{pref}), provide the best balance between composition quality and computation time. For even faster responses, our Fast Forward variant FF_{enf}^{pref} is the best choice, although the developer must be aware of its high failure rate. Both GP_{enf}^{pref} and FF_{enf}^{pref} improve quality by enforcing the use of all input parameters by the compositions they generate. If, on the other hand, tolarence to missing inputs is important, GP_{miss}^{pref} showed to be the most effective option – with high success rate and near 0.7 average quality – while keeping the computation time below one second.

In the previous Chapter we pointed out that there are other planning strategies that could be used to mimic the features implemented in our algorithms, namely, planning with preferences (enforced inputs) and conformant planning (missing inputs). For the evaluation in this Thesis, however, we were able to compare our algorithms to Classical Planning tools only, and even then, the evaluation encountered several obstacles due to the inability of the tools for handling large domains, as reported in Section 6.1.1. In fact, two of the planners we tested support soft preferences – SGPlan and MIPS-XXL –, but they either crashed (MIPS-XXL) or we could not reproduce the same behavior of our algorithms (SGPlan). However, as a thorough comparison could not be carried out on time for this Thesis, we cannot rule them out right away. This comparison is a subject to future work.

Chapter 7 Final Remarks

Automated Service Composition (ASC) has been studied for various years already and several proposals emerged during these years, with varied levels of maturity. Nonetheless, the use of automated composition techniques by service developers has not taken off yet, despite the efforts of the Web Services Community. In our opinion, this is due, in part, to the fact the some complex approaches seek to reduce the role of the developer in the (automated) composition process, while even trying to replace him/her altogether; other simpler approaches, conversely, may be too dependent on the developer, requiring constant feedback in order to find solutions. We think that automated composition can be an important item in the developer's utility belt – akin to code completion and refactoring in modern IDE's –, but for that to happen it must be simple to use, non-intrusive, relatively fast and more tolerant to human failures. Moreover, the developer should be able rely on the solutions provided by the algorithms, or at least have some expectation of how reliable (or not) the compositions generated automatically are.

In this Thesis, we have worked on some of the problems that hinder the applicability of automated composition in practical scenarios. We chose to work with simple Input/output-based composition specifications, which limit the amount of information that the developer needs to provide initially. We investigated the use of AI planning, the most recurring technique for automated composition, and measured its effectiveness. To circumvent the lack of semantic information that is common for real-life services, we opted for using a publicly available repository of scientific workflows, from which we extracted the information necessary to the planning algorithms. From the experience of using planners to solve the composition problems, we identified some major improvement points. To demonstrate the ideas, we enhanced two classic planning algorithms in order to increase the quality of their solutions, measured as the accuracy of the automated compositions compared to ones written manually. After performing experiments intended to validate our contributions, we have come to the following conclusions.

7.1. CONCLUSIONS

The conclusions of this Thesis are the following:

1. **Quality of the Solutions:** Automated composition using standard planning algorithms can indeed provide solutions with acceptable quality. It is possible, however, to increase the quality by applying planning algorithms specially crafted to the service composition task. In special, ensuring the adherence of the solution to the initial specification by enforcing the use of all input parameters showed capable of raising significantly the quality of the solutions.
2. **Incomplete Specifications:** The lack of complete knowledge of the developer regarding the composition specification can be tackled with adequate algorithms, hence improving the developer's productivity by offering approximate solutions in early phases of the development, when not all details are known. Our algorithms can handle the case when the specification lacks some input parameters, providing an approximate solution that uses the inputs provided and adds a configurable number of extra inputs necessary for the composition. Without such improvements, the developer would have to deal with "no solution found" responses from the planning algorithm until he/she finds out which input parameters are missing.
3. **Composition Time:** Using classical planning tools can be too time-consuming for agile development scenarios. Our experiments showed that a simplified, more focused implementation can be orders of magnitude faster than a generic planner, which suggests that expressive power may need to be scarified in favor of usability. Irrespective of the tool used, some problem instances will still require more time than the developer may be willing to wait, as timed-out cases happened to most algorithms (especially when trying to improve the solution quality). Our opinion based on the experience with planning algorithms – and shared by other authors [33] – is that they either find a solution "quickly" or hang indefinitely looking for it.
4. **Semantic Descriptions:** It is possible to employ automated composition techniques without manually written semantic annotations attached to services, as long as a composition repository of significant size is available. By applying a simple reverse engineering procedure to the repository, it is possible to obtain acceptable quality compositions automatically. Although the repository we used contains service compositions for scientific data processing instead of the more

common business-oriented domain, we believe that our process can be applied with minor changes to any repository of compositions, be it public or private.

7.2. CONTRIBUTIONS

The contributions of this Thesis are the following:

1. **Algorithms for improving the quality of the compositions.** The algorithms try to ensure that the solutions adhere to the specification given by the developer by making the compositions found use all the inputs parameters provided.
2. **Algorithms that tolerate the lack of information in the initial specification.** These algorithms increase the probability of finding a solution even if not all necessary input parameters are known beforehand while preserving the quality of the solutions at the same time;
3. **An evaluation of the proposed algorithms along with a state-of-the-art planning system.** The evaluation measured the quality of the solutions (accuracy) and overall performance of the tools. A real life composition repository was used in the evaluation, generating a planning domain large enough to break several state-of-the-art planners. In this evaluation, we measured not only the time taken for generating the solutions but also their quality. By measuring the quality of the solutions, we could identify problematic cases and adapt algorithms of the literature to handle them.
4. **A process for extracting semantic information from a database of service compositions.** The semantic data, comprised of compatibility information on the input and output parameters and also the precedence relation between services, enabled the use of automated composition techniques;

7.3. FUTURE WORK

In this Thesis, we focused on simpler, dataflow-oriented compositions where the main elements are services and the connections between them. Current composition languages, such as BPEL, allow for more elaborated compositions, resembling traditional programming languages. Control structures such as if's and for loops along with mechanisms to coordinate the parallel execution of services in the composition are available in these languages. In our opinion, however, these control structures would require much richer composition specifications than the basic input/output approach we used in order to make clear to the composition algorithms that such structures would be necessary at some point. The richer the specification, the more complex will be to the developer's job of describing the composition in the first place (not to mention that the underlying services would likely need more detailed descriptions as well), to the point that specifying the composition could eventually take the same effort as writing the composition itself. With that in mind, and for the purpose of having automated composition assisting – not replacing – the developer, having full-blown control-structures into solutions might be overkill. We want to investigate this subject further and verify to what extent these insights are valid or not.

The Input/output approach we adopted, however, can be too restrictive. Sometimes, for example, the developer is not sure about the outcome of the composition but knows some of the services that might be part of the final solution. We have already contemplated with this idea by proposing a graphical user interface where the developer can specify some services and leave the others for the composition algorithm to solve (SERVICES 2011 paper). We plan to elaborate on this idea so that the specification of the composition can be made as simple and flexible as possible.

The algorithms we developed can be subject to several improvements. Among them, we plan to add support to non-functional requirements, such as Quality of Service, which will give more control over the services used. We plan to improve the quality of the solutions, especially our FF variant, by reducing the error rate, i.e., number of wrong services included in the solutions (the number of correct services found – or hit rate – is already at a good level of 80%). Memory footprint of our FF variant is also an issue for further work since some instances failed due to lack of memory.

In the reverse engineering front, we plan to use ideas from theory of Social and Complex Networks [51] to mitigate the problem of low service reuse, which reduces the richness of information gathered from compositions and, consequently, the effectiveness of our automated composition approach. The overall approach would be to create new service relationships based on the existing ones, using for example, the link prediction techniques shown in [52]. The use of social aspects of the compositions – such as their authors and the services they use most frequently – can also contribute to enrich the service descriptions. Finally, we want to evaluate our reverse engineering process with other composition repositories.

Another interesting direction would be to measure effectively the time saved by applying automated composition. For that purpose, we would need to ask volunteers to implement some compositions manually and then fix the automatically generated compositions for the same specifications, measuring the time taken to perform both activities. In order to have a more meaningful quality measure, we could also ask the volunteers to rate the solutions found automatically – similar to Mean Opinion Score – and them find a formulation to map the composition attributes into the score given by the volunteers.

7.4. PUBLICATIONS

7.4.1. On Service Composition or Related Subjects

- In the following paper we show early results of our Graphplan-based approach for service composition and a proposal of Graphical User Interface that allowed both manual and automated composition:

  DANTAS, R., AZEVEDO, E., DIAS, C., LIMA, T., SADOK, D., KAMIENSKI, C. A., OHLMAN, B., "Facilitating Service Creation via Partial Specification and Automated Composition", In: Composition Workshop at the 2011 IEEE World Congress on Services (SERVICES), Washington, DC, 2011.

- We explore the combination of services and policies using a semi-automated service creation process called Service Refinement Cycle in the paper:

  KAMIENSKI, C. A., **DANTAS, R.,** AZEVEDO, E., DIAS, C., SADOK, D., OHLMAN, B., "Unleashing the power of policies for service-oriented computing", In: 7th International Conference on Network and Service Management (CNSM), Paris, 2011.

- This Book Chapter describes a proposal for combining P2P technologies and Policies in order to manage distributed services:

  KAMIENSKI, C. A., **DANTAS, R.,** SADOK, D., OHLMAN, B., "Managing the Future Internet: Services, Policies and Peers. In: Handbook of Research on P2P and Grid Systems for Service-Oriented Computing: Models, Methodologies and Applications. 1.", IGI Global, Hershey, EUA, 2010. **(Book Chapter)**

- In this Application Patent we present some of the ideas on the execution of policy-equipped services:

  KAMIENSKI, C., **DANTAS, R.,** OHLMAN, B., SADOK, D., "Method and Apparatus for the Execution of Adaptable Composed Computer-Implemented Services with Integrated Policies", U.S. Patent Application No. 12/969331, Unpublished (filing date December 5, 2010) **(Patent Application)**

- The paper that proposes the Service Refinement Cycle, a semi-automated framework for service creation:

  KAMIENSKI, C. A., **DANTAS, R.,** FIDALGO, J., SADOK, D., OHLMAN, B., "Service Creation and Execution with the Service Refinement Cycle", In: IEEE/IFIP Network Operations and Management Symposium (NOMS 2010), 2010, Osaka, 2010.

- An early work on the use of goals for the automatic creation of autonomic services:

  CHONG, M.-Y., BJURLING, B., **DANTAS, R.,** KAMIENSKI, C. A., OHLMAN, B., "GOAL-BASED Service Creation Using Autonomic Entities", In: Lecture Notes in Computer Science: Modelling Autonomic Communications Environments, Springer, Heidelberg, 2009.

7.4.2. Other Publications during the PhD

- **DANTAS, R.,** BERG, M., CEDERHOLM, D., GONCALVES, G., SIQUEIRA, R., RABELO, R., SADOK, D., "Method and arrangement in a communication system", WO Patent Application No. PCT/SE2010/000088, Unpublished (filing date April 1st, 2010) **(Patent Application)**

- GONCALVES, G., **DANTAS, R.**, PALHARES, A., KELNER, J., FIDALGO, J., SADOK, D., ALMEIDA, H., BERG, M., CEDERHOLM, D., "Estimating Video Quality over ADSL2+ under Impulsive Line Disturbance", In: Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering: AccessNets. Springer, Heidelberg, 2009.

- FIDALGO, J., KAMIENSKI, C. A., **DANTAS, R.,** SADOK, D., OHLMAN, B., "Policy Processing: Don't Take it for Granted", In: IEEE International Symposium on Policies for Distributed Systems and Networks, 2009, London. Proceedings of the International

Symposium on Policies for Distributed Systems and Networks. Los Alamitos, CA : IEEE Computer Society, 2009.

- GONCALVES, G., **DANTAS, R.,** PALHARES, A., KELNER, J., FIDALGO, J., SADOK, D.; ALMEIDA, H. ; BERG, M. ; CEDERHOLM, D., "Estimating Video Quality over ADSL2+ under Impulsive Line Disturbance", In: Third International Conference on Access Networks (AccessNets 2008), Las Vegas, 2008.

- **DANTAS, R.,** FIDALGO, J., SADOK, D., KAMIENSKI, C. A., OHLMAN, B., "Policies for the Management of Ambient Networks: From Theory to Practice", In: IEEE Workshop on Policies for Distributed Systems and Networks, New York, 2008.

- BERGLUND, A., BJURLING, B., **DANTAS, R.,** ENGBERG, S., GIAMBIAGI, P., OHLMAN, B., "Toward Goal-Based Autonomic Networking", In: IEEE Workshop on Distributed Autonomous Network Management Systems (DANMS'08), New Orleans, 2008.

- **DANTAS, R.;** FIDALGO, J. ; LIMA, J. S. M. ; SADOK, D. ; KAMIENSKI, C. A. ; OHLMAN, B., Effective Implementation of Network Composition for Ambient Networks. In: IEEE/IFIP Network Operations and Management Symposium (NOMS), Salvador-BA, 2008. **(Short paper)**

- KAMIENSKI, C. A.; FIDALGO, J.; **DANTAS, R.**; SADOK, D.; OHLMAN, B.; Design and Implementation of a Policy-based Management Framework for Ambient Networks: Choices and Lessons Learned. In: IEEE/IFIP Network Operations and Management Symposium (NOMS), Salvador-BA, 2008. **(Short paper)**

References

[1] AGARWAL, V., CHAFLE, G., DASGUPTA, K., KARNIK, N., KUMAR, A., MITTAL, S., SRIVASTAVA, B., "Synthy: A system for end to end composition of web services," Web Semantics vol. 3, issue 4, pp. 311-339, Elsevier, December 2005.

[2] AKKIRAJU, R., SRIVASTAVA, B., IVAN, A., GOODWIN, R., SYEDA-MAHMOOD, T., "SEMAPLAN: Combining planning with semantic matching to achieve web service composition," IEEE International Conference on Web Services, pp. 37-44, 2006.

[3] Amazon Simple Workflow Service, http://aws.amazon.com/swf/

[4] ANKOLEKAR, A., BURSTEIN, M., HOBBS, J. R., LASSILA, O., MARTIN, D. L., MCDERMOTT, D., MCILRAITH, S. A., NARAYANAN, S., PAOLUCCI, M., PAYNE, T. R. AND SYCARA, K., "DAML-S: Web Service Description for the Semantic Web," International Semantic Web Conference (ISWC), Sardinia, Italy, June 2002.

[5] ARPINAR, I. B., ALEMAN-MEZA, B., ZHANG, R., MADUKO, A., "Ontology-driven web services composition platform," IEEE International Conference on E-Commerce Technology, pp. 146-152, 2004.

[6] BAIER, J. A., MCILRAITH, S. A., "Planning with Preferences", AI Magazine, Vol 29, No 4, 2009.

[7] BASTIAN, M., HEYMANN, S., JACOMY, M., "Gephi: An Open Source Software for Exploring and Manipulating Networks", International AAAI Conference on Weblogs and Social Media, 2009.

[8] BERARDI, D., CALVANESE, D., DE GIACOMO, G., LENZERINI, M., MECELLA, M., "Automatic Composition of e- Services," 1st Int. Conf. on Service Oriented Computing (ICSOC 2003), Lecture Notes in Computer Science, vol. 2910, Springer, pp. 43-58, 2003.

[9] BERNERS-LEE, T., HENDLER, J., LASSILA, O., "The Semantic Web," Scientific American 284, May 2001.

[10] Biocatalogue: http://www.biocatalogue.org/. Last accessed in December 2011.

[11] BLUM, A., FURST, M., "Fast Planning Through Planning Graph Analysis", Artificial Intelligence, 90:281--300, 1997.

[12] BOLIE, J., CARDELLA, M., BLANVALET, S., JURIC, M., CAREY, S., CHANDRAN, P., COENE, Y., GEMINIUC, K., "BPEL Cookbook: Best Practices for SOA-based Integration and Composite Applications Development", Packt Publishing, 2006, ISBN: 1-904-81133-7.

[13] COLES A., SMITH, A., "Marvin: a heuristic search planner with online macro-action learning". *J. Artif. Int. Res.* 28, 1 (February 2007), 119-156, 2007.

[14] COLES, A. J., COLES, A. I., CLARK, A., GILMORE, S. T.,"Cost-Sensitive Concurrent Planning under Duration Uncertainty for Service Level Agreements", International Conference on Automated Planning and Scheduling (ICAPS-11), June, 2011.

[15] CORCHO, O., GÓMEZ-PÉREZ, A., "A Roadmap to Ontology Specification Languages". Polytechnic University of Madrid, 2000.

[16] CURBERA, F., DUFTLER, M., KHALAF, R., NAGY, W., MUKHI, N., WEERAWARANA, S., "Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI", IEEE Internet Computing, March/April 2002.

[17] DAML Project, http://www.daml.org/. Accessed in Nov. 2008.

[18] DOMSHLAK, C., HELMERT, M., KARPAS, E., MARKOVITCH, S., "The SelMax Planner: Online Learning for Speeding up Optimal Planning" (planner abstract). In Seventh International Planning Competition (IPC 2011), Deterministic Part, pp. 108-112. 2011.

[19] EDELKAMP, S., JABBAR, S., "MIPS-XXL: Featuring External Shortest Path Search for Sequential Optimal Plans and External Branch-And-Bound for Optimal Net Benefit", In 6th International Planning Competition Booklet, Sydney, Australia, Sept. 2008.

[20] ERL, T., "SOA Principles of Service Design", Prentice Hall PTR, 2007, ISBN: 0-132-34482-3.

[21] FARQUHAR, A., FIKES, R., RICE, J., "The Ontolingua Server: A Tool for Collaborative Ontology Construction". Proceedings of KAW96. Banff, Canada, 1996.

[22] FIKES, R.E., NILSSON N.J., "Strips: A new approach to the application of theorem proving to problem solving", Artificial Intelligence, 2 (3-4), pp. 189-208, 1971.

[23] GEREVINI, A., LONG, D., "Plan constraints and preferences in PDDL3", The International Conference on Automated Planning & Scheduling (ICAPS), 2006.

[24] GHALLAB M. ET AL., "PDDL: The Planning Domain Definition Language, Version 1.2", Technical Report CVC TR–98–003/DCS TR–1165, Yale Center for Computational Vision and Control, Yale Univ., New Haven, Conn., 1998

[25] GHALLAB, M., NAU, D., TRAVERSO, P., "Automated Planning: Theory & Practice", Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[26] GOBLE, C.A., BHAGAT, J., ALEKSEJEVS, S., CRUICKSHANK, D., MICHAELIDES, D., NEWMAN, D., BORKUM, M., BECHHOFER, S., ROOS, M., LI, P., DE ROURE, D.: "myExperiment: a repository and social network for the sharing of bioinformatics workflows", Nucl. Acids Res., 2010.

[27] GORTMAKER, J., JANSSEN, M., & WAGENAAR, R. W., "The Advantages of Web Service Orchestration in Perspective". 6th International Conference of Electronic Commerce, ICEC 2004.

[28] GRUBER, T. R., "A Translation Approach to Portable Ontology Specifications," Knowledge Acquisition, vol. 5, issue 2, June 1993.

[29] GUERRERO, A, VERGARA, J. E. L. DE, SÁNCHEZ-MACIÁN, A. & BERROCAL, J., "Ontology-based Policy Refinement Using SWRL Rules for Management Information Definitions in OWL", 17th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2006), October 2006.

[30] GUIZZARDI, G., WAGNER, G., "Some Applications of a Unified Foundational Ontology ins Business Modeling," Idea Group, 2005.

[31] HALLER, A., CIMPIAN, E., MOCAN, A., OREN, E., BUSSLER, C., "WSMX - A semantic service-oriented architecture," IEEE International Conference on Web Services, pp. 321-328, 2005.

[32] HELMERT, M., "The Fast Downward Planning System", Journal of Artificial Intelligence Research 26, 2006.

[33] HELMERT, M., RÖGER, G., SEIPP, J., KARPAS, E., HOFFMANN, J., KEYDER, E., NISSIM, R., RICHTER, S., WESTPHAL, M., "Fast Downward Stone Soup"

(planner abstract). In Seventh International Planning Competition (IPC 2011), Deterministic Part, pp. 38-45. 2011.

[34] HOFFMANN, J., NEBEL, B., "The FF Planning System: Fast Plan Generation Through Heuristic Search", Journal of Artificial Intelligence Research, Volume 14, pages 253 - 302, 2001.

[35] HOFFMANN, J., BERTOLI, P., PISTORE, M., "Web Service Composition as Planning, Revisited: In Between Background Theories and Initial State Uncertainty," 22nd National Conference of the American Association for Artificial Intelligence (AAAI'07), Vancouver, Canada, July 2007.

[36] Horn Rule Semantics Wiki, http://www.w3.org/2005/rules/wg/wiki/Horn_Rules_Semantics, Accessed in Nov. 2008.

[37] HORROCKS, I., PATEL-SCHNEIDER, P., "Reducing OWL Entailment to Description Logic Satisfiability," Journal of Web Semantics, 2004.

[38] HSU, C. W. , WAH, B. W., HUANG, R., CHEN, Y. X., "Handling Soft Constraints and Preferences in SGPlan", International Conference on Automated Planning and Scheduling, June 2006.

[39] HULL, D., WOLSTENCROFT, K., STEVENS, R., GOBLE, C., POCOCK, M., LI, P., OINN, T., "Taverna: a tool for building and running workflows of services.," Nucleic Acids Research, vol. 34, iss. Web Server issue, pp. 729-732, 2006.

[40] International Planning Competition, http://ipc.icaps-conference.org/.

[41] KAUTZ, H., SELMAN, B., "BLACKBOX: A New Approach to the Application of Theorem Proving to Problem Solving", Working notes of the Workshop on Planning as Combinatorial Search, held in conjunction with AIPS-98, Pittsburgh, PA, 1998.

[42] KAUTZ, H., SELMAN, B., HOFFMANN, J., "SatPlan: Planning as Satisfiability", Abstracts of the 5th International Planning Competition, 2006.

[43] KHOSHAFIAN, S., "Service Oriented Enterprises," Auerbach, 2006.

[44] KIFER, M., LAUSEN, G., WU, J., "Logical Foundations of Object-Oriented and Frame-Based Languages," Journal of the ACM. 1995.

[45] KLUSCH, M., FRIES, B., SYCARA, K., "Automated semantic web service discovery with OWLS-MX," International Joint conference on Autonomous Agents and Multiagent Systems (AAMAS '06), pp. 915-922, 2006.

[46] KLUSCH, M., GERBER, A., "Semantic web service composition planning with OWLS-XPlan", 1st Int. AAAI Fall Symposium on Agents and the Semantic Web, Arlington VA, 2005.

[47] Knowledge Interchange Format draft proposed American National Standard: http://logic.stanford.edu/kif/dpans.html. Accessed in Nov. 2008.

[48] KOEHLER, J., "IPP - A Planning System for ADL and Resource-Constrained Planning Problems", Habiliation Thesis, University of Freiburg, 1999.

[49] LAUSEN, H., DE BRUIJN, J., POLLERES, A., FENSEL, D., "WSML - a Language Framework for Semantic Web Services," W3C Workshop on Rule Languages for Interoperability, Washington DC, USA, 2005.

[50] LÉCUÉ, F., LEGER, A., "Semantic web service composition through a matchmaking of domain,". European Conference on Web Services, pp. 171-180, 2006.

[51] LEWIS, T. G., "Network Science: Theory and Applications", John Wiley and Sons, Mar 2009.

[52] LIBEN-NOWELL, D., KLEINBERG, J., "The link prediction problem for social networks". In Proceedings of the twelfth international conference on Information and knowledge management (CIKM '03). ACM, New York, NY, USA, 2003.

[53] LIN, N., KUTER, U., SIRIN, E., "Web service composition with user preferences", 5th European Semantic Web Conference (ESWC'08), Springer-Verlag, Berlin, Heidelberg, pp. 629-643, 2008.

[54] MACGREGOR, R., "Inside the LOOM clasifier". SIGART bulletin. #2(3):70-76. June, 1991.

[55] MCCANDLESS, D., OBRST, L., HAWTHORNE, S., "Dynamic web service assembly using owl and a theorem prover," International Conference on Semantic Computing, pp. 336-341, 2009 IEEE International Conference on Semantic Computing, 2009.

[56] MCILRAITH, S. A., SON, T. C., ZENG, H., "Semantic Web services," IEEE Intelligent Systems, vol. 16, no. 2, pp. 46- 53, Mar-Apr 2001.

[57] MCILRAITH, S., SON, T., "Adapting Golog for Composition of Semantic Web Services," International Conference on Knowledge Representation and Reasoning (KR2002), pp. 482-493, Toulouse, France, April 2002.

[58] MINSKY, M., "A framework for representing knowledge," The Psychology of Computer Vision. New York, McGraw-Hill, 1975.

[59] MOTTA, E., "Reusable Components for Knowledge Modelling". IOS Press. Amsterdam. 1999.

[60] myExperiment: http://www.myexperiment.org/. Last accessed in December 2011.

[61] myGrid, http://www.mygrid.org.uk/. Last accessed in December 2011.

[62] NAU, D., AU, T-C., ILGHAMI, O., KUTER, U., MURDOCK, J. W., WU, D., YAMAN, F., "SHOP2: an HTN planning system," Journal of Artificial Intelligence Research, vol. 20, issue 1, pp. 379-404, December 2003.

[63] OASIS, "Reference Model for Service Oriented Architecture 1.0", August 2006.

[64] OASIS, "Web Services Distributed Management: Management Using Web Service (MUWS 1.0) Part 1." Available at http://docs.oasis-open.org/wsdm/2004/12/wsdm-**muws**-part1-1.0.pdf. Accessed in Nov. 2008.

[65] OASIS, "WS-BPEL 2.0", http://docs.oasis-open.org/wsbpel/2.0/OS/ wsbpel-v2.0-OS.html. Accessed in Nov. 2008.

[66] OMER, A. M., SCHILL, A., "Web service composition using input/output dependency matrix," 3rd Workshop on Agent-Oriented Software Engineering Challenges for Ubiquitous and Pervasive Computing, ACM, pp. 21-26, London, United Kingdom, 2009.

[67] OWLS-TC – OWL-S test collection, http://www.semwebcentral.org/projects/owls-tc/.

[68] Parlay X, http://www.parlayx.com/.

[69] PDDL4J Library, http://sourceforge.net/projects/pdd4j/

[70] PELTZ, C., "Web Services Orchestration and Choreography", Computer Magazine IEEE, 2003.

[71] Prover9 and Mace4, http://www.cs.unm.edu/~mccune/prover9/.

[72] REITER, R., "Knowledge In Action: Logical Foundations For Specifying And Implementing Dynamical Systems", The MIT Press, 2001.

[73] RIBIÈRE, M, CHARLTON, P., "Ontology Overviews". Motolora Labs, France.

[74] RICHTER, S., WESTPHAL, M., HELMER, M., "LAMA 2008 and 2011" (planner abstract). In Seventh International Planning Competition (IPC 2011), Deterministic Part, pp. 50-54. 2011.

[75] RODRÍGUEZ-MIER, P., MUCIENTES, M., LAMA, M., COUTO, M., "Composition of web services through genetic programming," Evolutionary Intelligence, Springer, pp. 171-186, 2010.

[76] SCHITTKO, C., "Web Service Orchestration with BPEL", XML Conference and Exposition, December 2003.

[77] SINGH, M. P., HUHNS, M. N., "Service-Oriented Computing – Semantics, Process, Agents", John Wiley & Sons, 2005, ISBN 0-470-09148-7.

[78] SOHRABI, S., PROKOSHYNA, N., MCILRAITH, S. A., "Web Service Composition via Generic Procedures and Customizing User Preferences", 5th International Semantic Web Conference, Athens GA, 2006.

[79] TAN, W., ZHANG, J., MADDURI, R., FOSTER, I., DE ROURE, D., GOBLE, C., "ServiceMap: Providing Map and GPS Assistance to Service Composition in Bioinformatics", 2011 IEEE International Conference on Services Computing (SCC); Washington, DC, 4-9 July 2011.

[80] TAN, W., ZHANG, J., FOSTER, I., "Network Analysis of Scientific Workflows: A Gateway to Reuse", IEEE Computer; Volume: 43 Issue:9; Sept. 2010.

[81] Taverna Workbench: http://www.taverna.org.uk/. Last accessed in December 2011.

[82] The Rule Markup Language Initiative, http://www.ruleml.org/. Accessed in Nov. 2008.

[83] TRAVERSO, P., PISTORE, M., "Automated Composition of Semantic Web Services into Executable Processes", International Semantic Web Conference (ISWC), Hiroshima, 2004.

[84] VAQUERO, L. M., RODERO-MERINO, L., CACERES, J., LINDNER, M., "A break in the clouds: towards a cloud definition," *SIGCOMM Computer Communications Review,* vol. 39, issue 1, pp. 50-55, December 2008.

[85] VERGARA, J. E. L. DE, VILLAGRÁ, V. A. & BERROCAL, J., "Benefits of Using Ontologies in the Management of High Speed Networks", Lecture Notes in Computer Science, Springer – 2004.

[86] W3C Consortium, "Web Services Choreography Description Language Version 1.0", http://www.w3.org/TR/ws-cdl-10/. Accessed in Nov. 2008.

[87] W3C Section - SWRL: A Semantic Web Rule Language Combining OWL and RuleML. http://www.w3.org/Submission/SWRL. Accessed in Nov. 2008.

[88] W3C Web Ontology Language Overview. http://www.w3.org/TR/owl-features. Accessed in Nov. 2008.

[89] W3C, "OWL-S: Semantic Markup for Web Services," http://www.w3.org/Submission/OWL-S/. Accessed in Nov. 2008.

[90] Web Services Challenge 2008, http://cec2008.cs.georgetown.edu/wsc08/index.html.

[91] WONG, A. K. Y., RAY, P., PARAMESWARAN, N. & STRASSNER, J., "Ontology Mapping for the Interoperability Problem in Network Management", IEEE Journal on Selected Areas in Communications, vol 23, no. 10, October 2005.

[92] WU, D., PARSIA, B., SIRIN, E., HENDLER, J., NAU, D., "Automating DAML-S web services composition using SHOP2," Lecture Notes in Computer Science, Springer Berlin, pp. 195-210, vol. 2870, 2003.

[93] XIAO, D. & XU, H., "An Integration of Ontology-based and Policy-based Network Management for Automation", IEEE International Conference on Computational Intelligence for Modelling Control and Automation, and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06).

[94] YAHIA, I. G. B., BERTIN, E. & CRESPI, N., "Ontology-based Management Systems for the Next Generation Services: State-of-the-Art", Third International Conference on Networking and Services(ICNS'07).

[95] YAN, Y., ZHENG, X., "A Planning Graph Based Algorithm for Semantic Web Service Composition", 10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services (CECANDEEE '08), IEEE Computer Society, Washington, DC, 2008.

[96] ZHANG, J., TAN, W., ALEXANDER, J., FOSTER, I., MADDURI, R., "Recommend-As-You-Go: A Novel Approach Supporting Services-Oriented Scientific Workflow Reuse", 2011 IEEE International Conference on Services Computing (SCC), Washington, DC, 4-9 July 2011.

[97] ZHANG, R., ARPINAR, I. B., ALEMAN-MEZA, B., "Automatic composition of semantic web services,", International Conference on Web Services, pp. 38-41, Las Vegas, 2003.