Context Verification and Adaptation in Web Service Composition

Touraj Laleh

A Thesis

In the Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of

Doctor of Philosophy (Computer Science) at

Concordia University

Montréal, Québec, Canada

January 2018

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: Touraj Laleh

Entitled: Context Verification and Adaptation in Web Service Composition

and submitted in partial fulfillment of the requirements for the degree of

Doctor Of Philosophy (Computer Science)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Anjali Awasthi — Chair

Dr. Gilbert Babin — External Examiner

Dr. Roch H. Glitho — External to Program

Dr. Volker Haarslev — Examiner

Dr. Rene Witte — Examiner

Dr. Joey Paquet — Thesis Co-Supervisor

Dr. Yuhong Yan — Thesis Co-Supervisor

Approved by _____

Dr. Volker Haarslev, Graduate Program Director

Monday, February 5, 2018

_____

Dr. Amir Asif, Dean

Faculty of Engineering and Computer Science

Abstract

Context Verification and Adaptation in Web Service Composition

Touraj Laleh, Ph.D.

Concordia University, 2018

Automatic web-service composition aims at automating the design of an appropriate combination of existing web services to achieve a global goal. Most proposed AWSC approaches only consider input/output parameters and quality features of services. However, most real-world web services have applicable conditions and require constraints to be considered according to the execution context of composite services. Constraint verification has a significant impact on the composition and execution of composite services. In particular, run time verification of service constraints can result in the failure of the execution of composite services and eventually waste computational resources and may incur monetary costs. In addition, traditional adaptation approaches for web service composition consider recovery in case of failure when a service becomes unavailable. They do not take into account changes and limitations in service execution environment which potentially can affect the execution of a wide range of services. Externally-defined constraints are likely to be defined and become or cease to be applicable after the composite service has been deployed. In this thesis, we propose a novel approach to model and verify different types of constraints inside composite services. We not only consider input/output parameters but also the values that can be assigned to parameters during design and execution of composite services. In addition, we provide novel failure recovery and adaptation approaches for different types of constraints according to the execution context of composite services. In our solution, we develop a new structure including alternative composite services to recover broken composite services and adapt to

external constraints. We finally propose a brokerage architecture including all proposed approaches for constraint-aware service composition and adaptation.

Acknowledgments

There have been many people who have walked alongside me during the last five years. They have guided me and motivated me to finish this research. I gratefully express my deepest gratitude to my supervisor Dr. Joey Paquet for his great supervision, support and valuable advice without which this thesis would not have been possible. I am also grateful to Dr. Yuhong Yan and Dr. Serguei A. Mokhov for their support and valuable advice.

I would also like to extend my thanks to all of my dear friends, specially, Ata, Soli, Mojtaba, Reza, Amo Hadi, Mostafa, Elaheh, Mehdi, Arash, Kasra, Hamed, Hassan, Alex, Peyman, Sleiman and Jyotsana whose pure friendship has motivated my social and academic life in Canada.

Finally and without hesitation I would like to thank my mother to whom this thesis is dedicated for her belief in me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In the highly competitive environment of the web, enterprises can use service providers to fulfill their business goals while themselves focusing on their core business. Service-oriented architecture (SOA) is an architectural pattern in which application components provide on-demand software systems to other components [74]. SOA helps lower the costs of software development and management compared to the traditional software development paradigm that implements new systems from scratch. SOA follows the find-bind-execute paradigm (Figure 1) in which:

- Service Requester is the client who searches for desired services.
- Service Provider implements and provides functionalities as web services. It also registers service information in a service registry. The service operations and interface can be described in WSDL (Web Service Description Language) documents.
- Service Registry who publishes services in a service repository.

Web services are self-contained, self-describing, modular applications that represent functionalities on the web, and can be invoked across the web [79]. In most researches [83, 48], each service is represented by a set of features including

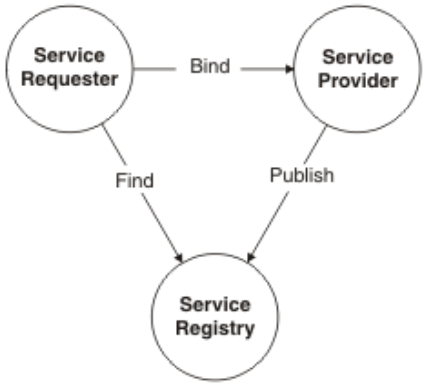


Figure 1: Service-oriented architecture paradigm.

input/output and quality features. After a service has been published, it should be able to be discovered and selected by service requesters. Then, it could be executed several times by service users. This problem is referred to as the web service selection problem which has been discussed in many researches [87, 4, 3]. They discuss different methods to select a service based on functional capabilities and quality of service. Quality of Service (QoS) refers to the non-functional properties of services such as availability, response time, throughput, cost, execution duration, reputation and successful execution rate. However, the service selection process might not be able to select a single service to accomplish a more complex task. In this situation, service requesters may fulfill their complex business requirements by combining different web services. Web service composition tries to find a chain of connected services in which output parameters of a service are used as part of input parameters of another service (or a group of services). Automatic web-service composition (AWSC) consists in the automated assembly of an appropriate combination of existing web services to achieve a global goal. Many approaches have been proposed for AWSC problem using different domains such as artificial intelligence [69, 107] and formal methods [47, 10]. We demonstrate a scenario of a shopping plan to show the service composition motivation:

- A service requester searches for a service to do online shopping. The shopping service should be able to do the following tasks: searching for products,

submitting an order, paying for the order, and shipping/delivery of the order. As no individual web service provides all required tasks, a composite service needs to be constructed by combining different services.

- Services for different tasks (searching, ordering, ...) are chosen from the service repository and a composition algorithm constructs a searching graph from initial states to the goal with chosen services.

- Based on the service requesters' constraints, a composite service solution is extracted from the searching graph.

- Services are combined as a new service, which is made available as an executable service to service users.

Another factor that needs to be considered in web service composition is that web services are not universally applicable. It means, they can not be executed in all contexts. Although many researches have been done on using context-aware computing in different domains, the notion of a "context" is somewhat vague to define in different areas including web service environment. We define context as any information that affects the execution of a service. During the execution, a service gets values assigned to input parameters from the context and modifies values of output parameters in the context. Context information could relate to service requester, service providers, service users, and execution environment of the service. Before executing a service in a certain context, it should be verified whether the service can be executed properly in such context or not. *Constraints* express limitations that can be used to verify whether a service can be executed in a certain context. They express conditions or restrictions, which are imposed by service requesters, service providers, service users and execution environment. For example, a web service to do the shipment may have a constraint only to enable it to ship items to/from North America. Therefore, for any item that needs to be shipped using this service, this constraint should be verified in advance and if the pickup or delivery address is not in North America, the service cannot be executed. Another real-world example is

GlobalWeather Service [65] which provides weather reports for a list of major cities around the world. Clearly, this service is not providing the weather report for every city in the word, which is a constraint of GlobalWeather Service.

For a composite service, the set of constraints is derived from the constraints of the services inside the composite plan [27]. For example, consider a composite travel booking service including *GlobalWeather* service and a hotel booking service. The composite service gets a city name and a booking date as input and returns the weather report and the list of all hotels of the city. It is clear that the travel booking service can only return proper results for cities whose weather can be checked by *GlobalWeather* service. Therefore, it could be said that the constraints of the composite service are derived from *GlobalWeather* service. We call these types of constraints *internal constraints* or *service constraints* of a composite service as they belong to providers of services inside the plan.

Internal constraints are not the only constraints that need to be considered at execution time of a composite service. Even after the initial assembly and deployment of the composite service, new constraints might be imposed on the composite service and services inside the plan. For example, consider the above-mentioned shipment service (which only ships items from/to North America) is used in a shopping composite service. After the service has been composed and deployed, new international regulations might come to stop the clients from shipping any item to/from the United States. Additionally, such a constraint might also be lifted/re-applied later in the future. This requirement imposes new constraints on the shopping composite service. Considering the shipment service constraints (ship only to/from North America) and the newly added constraint (not to ship to/from the United States) makes the composite service to only let the users to shop to/from any address in North America except the United States. These types of constraints are not limitations given by a service provider. They come externally and they need to be applicable after the composite service has been assembled and deployed, i.e., it requires dynamic injection of the constraints in the composite service. We call these limitations *external constraints*, compared to internal constraints which are

only defined by service providers.

1.2 Problem Statement

At execution time of a composite service, constraints of services inside the plan must be verified before their execution inside the composite plan. If verification of a service constraint of a service inside the composite plan fails, it will result in execution failure of the composition plan. In this situation, all service executions that have been done in the composition plan up to the failure point will potentially have to be rolled back which will also result in wastage of computational resources. It should be noted that it is not possible to entirely avoid execution failures of composite services as the constraint verification of services inside the plan might depend on the execution results of other services.

In addition, individual execution of a composite service for a single task might not be complete as a result of a failure in verification of service constraints during the execution of a composite service. Therefore, a failure recovery approach needs to be employed to recover the composition plan and finish execution of the broken task. Current failure recovery approaches are not efficient to recover constraint verification failures and minimize wasting of time and computational resources. There is a difference between failure recovery resulting from the unavailability of services inside the plan and failure resulting from service constraint verification. When a service from services inside the plan is not available, it will be excluded from a composite service plan by the recovery process and any plan using the service will no longer be valid. However, when service constraint verification fails in the execution of the composite service, the recovery process should find an alternative plan to complete the execution. This should also take into consideration that the plan may still be valid for some following executions, hence it is not required to be excluded from available plans. Therefore, going through the recovery process for every verification failure inside the composite service could add considerable overhead to system performance during the execution of the composite service [44].

In a highly dynamic environment, external constraints might be defined and applied dynamically and composite services should be able to adapt to them immediately when they apply, and similarly adapt to the removal of the constraints when they cease to apply. Current web service composition adaptation approaches have some drawbacks. First, they usually require time costs similar to the original planning process needed. Second, they abandon some parts of the existing plan (even the whole plan in some cases). Therefore, the resulting plan can be a very different plan with new services and specifications from its predecessor. This might not be acceptable in the real world since users often sign contracts with web services providers, which is defined according to the original plan. Changing the composition plan means to redefine, renegotiate, or often abandon existing business contracts. Additionally, changing a composite service plan will most often result in changing the data model used by a composite service, which might be highly problematic if a composite service is expected to retain all transactional data of service usage.

1.3 Thesis Objectives

Current constraint verification approaches might result in wastage of computational resources. In fact, some services of a composite service become unusable as some of their constraints are not satisfied when they are placed in a particular context of execution. In our **first objective**, we are motivated to keep track the context and the constraints and verify all related constraints as soon as the context changes. In this way, the upcoming failure can be caught sooner and the number of rollback penalties could be minimized. However, to track the context and the constraints and verify all related constraints, we need to identify dependencies among services in a composite service. We are motivated to formally express dependencies among component services in a chain of composite services and propose mechanisms to avoid unnecessary executions. To do that, we first define a model including different concepts such as service and constraint. Then considering drawbacks of current constraint verification approaches, we focus on solving the web service composition problem as well as the

constraint verification problem in web service composition. We are motivated to use the ability of our proposed model in expressing service constraints and dependencies among services in a chain of composite services to improve the effectiveness of the verification process and minimize wastage of computational resources.

As we discussed in Section 1.1 constraint verification can result in the execution failure of the composite service, which requires failure recovery of the composite service plan. In our **second objective**, we are motivated to design a novel approach for constraint failure recovery in web service composition. When service constraint verification fails a single execution of the composite service, our proposed approach should find an alternative plan to complete the composite service execution, taking into consideration that the plan is still valid for the following executions and it does not need to exclude the service from composite plans. Therefore, unlike current constraint verification approaches, it does not recover with a new plan to improve system performance.

In the **third objective**, we consider effects of externally defined constraints (not service constraints) on executing composite services. We argue that adaptation to external constraints does not necessarily require changes in the plan of a composite service. Therefore, we are motivated to propose a new structure for composite services in which adaptation to new external constraints does not necessitate dealing with the re-construction of the composite service. We are also motivated to design required mechanisms to add/remove external constraints to/form composite services. In this way, to add an external constraint to a composite service, it is not necessary to find an alternative service with required constraints and reconstruct the plan. Therefore, adaptation performance can be improved as we do not need to go through the recovery process for every adaptation process.

1.4 Thesis Contributions

Considering the issues and objectives discussed in Section 1.3 the main contributions of this thesis are:

A constraint-aware service model and composition algorithm

The first requirement to address all discussed issues is to have a clear understanding of all related concepts. We developed a model for web service composition to express all required concepts formally. We also proposed algorithms based on planning-graph to generate constraint-aware composite services.

An efficient constraint verification method for web service composition

A novel constraint verification approach is designed to adjust the verification point of service constraints inside a constraint-aware composite service. It can reduce the cost of possible rollbacks necessitated by the execution failure of individual services. In our approach, we provide a method to model dependencies among services inside a composite service. Then, we move back the verification point of constraints inside a composition plan to avoid unnecessary execution of services in case of failure.

A failure recovery approach for constraint-aware composite service

Using the proposed efficient constraint verification method, an upcoming failure during the execution of composite service can be caught faster. A failure recovery approach is proposed to start recovery as soon as the upcoming constraint verification failure is caught during execution of composite services. In our proposed approach, we focus on defining a constraint-aware composite service package (CaCSP) which is a novel structure including different solutions for a composite service request to recover failure at execution time.

An adaptation approach for externally-defined constraints in web service composition

External constraints are formally defined and a solution to adapt a working composite service to external constraints is provided. Our adaptation approach can add/remove external constraints inside a composite service without reconstruction of the plan. Therefore, compared to current approaches, the adaptation time performance is significantly improved.

Policy-based Composite Package

In the real world, internal and external constraints must be considered in design and execution of composite services. Internal constraints need to be verified and adjusted.

External constraints should be added to composite plans and apply required effects. In our last contribution, we propose *Policy-based Composite Package* which is our proposed structure to handle different aspects related to verification, failure recovery and adaptation of internal and external constraints at runtime. We also propose an architecture of a *context/constraint-aware service brokerage* to manage creation and execution of policy-based composite packages.

1.5 Research Method

As we present different contributions of our research in a chapter, for each chapter we have a methodology section. In this section, we discuss the overview and design of the research problem and our proposed evaluation method. Figure 2, depicts our research methodology in each chapter. In each chapter, we start with an overview of the problem and then state the problem using a motivation example. Then, we formally define the required concepts and discuss required algorithms to represent our solutions. Finally, to be able to evaluate our solutions effectively, we provide mathematical and/or experimental evaluations. For experimental evaluation, we use a publicly available data set generator.

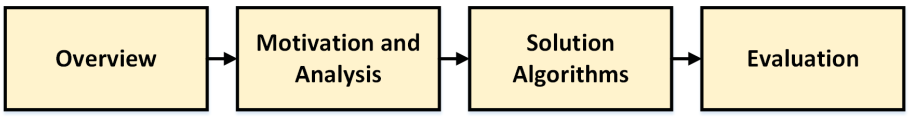


Figure 2: Research method.

1.6 Research Limitations

During our research, we made some assumptions that represent limitations of our work from theoretical and implementation perspectives. We discuss each limitation in detail as we present our solutions for different research problems in the following chapters. However, we discuss some of them in general as follow. First of all, we made an assumption that all services (atomic or composite) which could be represented

using different service descriptions can be translated using our proposed model. We made this assumption, because we did not want to focus on specific services representations in our model.

In addition, we used planning-graph approach which is a powerful and common method to solve the service composition problem. However, using this approach imposes some limitations on the generated composite services. First of all, there could not be a loop in the generated service composition plans. Second, there is no uncertainty in the execution of services. i.e. all services in a plan must be executed to make the execution of the plan complete. Third, a specific service cannot appear at multiple positions inside a composite service plan.

1.7 Thesis Organization

The thesis is organized as follows:

- **Chapter 2:** describes the preliminary knowledge and reviews of related works.
- **Chapter 3:** describes our proposed model and algorithms for web service composition and our novel constraint verification approach in web service composition.
- **Chapter 4:** proposes a novel recovery approach when verification of internal constraints fails execution of composite services.
- **Chapter 5:** proposes a novel adaptation approach to adapt composite services to externally-defined constraints.
- **Chapter 6:** proposes an approach and architecture to of a system which combines all proposed approaches.
- **Chapter 7:** describes the conclusion and future work of our research.

Chapter 2

Related Works

Web service composition is one the most critical paradigms in Service Oriented Architecture. It gained widespread popularity and has received the attention of many researchers and companies due to the promising benefits and challenges it offers.In this chapter, first, we discuss different web service composition approaches including graph-based and AI planning-based approaches. Then, we consider the role of context and constraints in the service environment and review current context/constraint-aware web service composition approaches. We finally discuss different web service composition adaptation and failure recovery methods and provide a comparison among recent approaches.

2.1 Web Service Composition

The automatic web service composition problem is addressed in many researches from different perspectives. One important aspect to discuss current service composition approaches is the number of solutions they can generate. Some methods only look for one possible solution among all solutions. These solutions generate only one possible plan [10, 11, 37]. Some methods generate a plan including generic templates of services [71, 92]. Among them, Yan et al. [107] propose an approach to find a solution (composite plan) for a web service composition problem in polynomial time, but with possible redundant services. In addition, some approaches look for more than

one possible solution or search for the optimal solution of a web service composition problem. Finding the optimal solution or all possible solutions (composition plans) for a web service composition problem is a well-known NP-complete problem [68, 92]. Many difficulties such as the huge search space, the identification and removal of redundant services, and the low efficiency of finding solutions restrict web service composition approaches that can generate all possible composite plans for a web service composition problem [71, 92]. In our research, we are interested in approaches that could create more than one possible solutions for a web service composition problem. According to the techniques, these approaches are mostly discussed in different categories such as formal methods-based approaches [47, 10], AI planning techniques [69, 107] and graph-search-based approaches [37, 92, 17].

Formal methods such as Petri net, finite state machine, and temporal logic have been used for modeling, verification and composition of web services. In [10] a finite state machine (FSM)- based framework for automatic service composition is developed, and effective techniques are provided for computing service composition where the behavioral description of a service is expressed as FSM. Brogi and Corfini [17] present a matchmaking system based on OWL-S and Petri nets for discovering deadlock-free compositions of web services. A global Petri net model is generated for a service registry through the data dependencies between services, and then the Petri net state equation technique is used to determine whether there is a composite service satisfying a request.

Due to their solid theoretical basis and rich tool support, formal method based approaches are mainly used for modeling and verification of web service composition. They allow to simulate and verify the behavior of a composition model at composition time to ensure correct expectations of a composite service according to the requirements and constraints of designers and planners. It has been argued that formal methods are often complicated and difficult to be implemented efficiently, thus are seldom used directly in the automatic web service composition.

Graph-based approaches usually construct a service dependency graph to show

all possible input/output dependencies among services in a registry. In most graph-based approaches the service dependency graph is a reflection of the underlying data interface relationships among services [92]. In this context, the automatic web service composition approach acts like a graph search problem and finds a path either from provided inputs to required outputs or vice-versa. Hashemian et al. [37] uses a modeling tool to convert the WSC problem into a general graph problem. First, they create a dependency graph contains information about existing web services in the repository. Then, composition solution plans could be found in the dependency graph based on web services that can potentially participate in the composition. Lang et al. [51] propose a solution to represent search dependency graph based on AND/OR graph where only one composite service template can be generated by their algorithm, which is computationally easier. An AND/OR graph can be seen as a generalization of a directed graph, and it is commonly used in automatic problem solving and problem decomposition. It contains two kinds of nodes: AND/OR, and they are connected by generalized edges called connectors. Wang et al. [92, 93] propose a formal graph-based service composition method based on AND/OR graph to find all possible solutions for a web service composition problem. They adopt AND/OR graphs for representation of search dependency graph. In this representation of the graph, a service can be executed only when all of its data nodes, which have AND among and are connected to this service node directly, are available. In contrast, there is a logical or relationship among those service nodes that can produce a certain data parameter because any one of them can generate this output and make the parameter available. Thus, all the data nodes in this representation are OR nodes. Finally, they present a search algorithm based on AND/OR graphs to find all the feasible composite service solutions based on this representation of search dependency graph.

Planning graph, an AI (Artificial Intelligence) algorithm, is a powerful data structure based on Planning Graph analysis for reaching a goal state. In this approach, given an initial state and a goal state, a sequence of actions can be acquired automatically through planning [79]. This approach is done in two stages: a forward expansion stage constructs a search graph and a backward searching stage retrieves

Table 1: Available services

Service	Input Parameters	Output Parameters
W_1	A,B	D
W_2	B,C	E
W_3	C,D	E
W_4	E	F

a solution [50]. The forward expansion stage builds the planning graph from the initial state. It loops over the service repository and adds available services into the planning graph. This process ends when no more services can be added to the action layer of the planning graph [15]. If the goal is contained in the action layer, there is at least one solution that can reach to goal state from initial parameters and then the backward searching approach is used to retrieve solutions from the planning graph. To make it more clear, Figure 3 depicts the planning graph based on a set of services in Table 1, in which the input and output parameters of web service composition problem are $I = A, B, C$ and the goal is $O = f$. This graph contains two layers including proposition layers and action layers contain services. First, input parameters of the web service composition problem will be added into P_0, and then the algorithm searches the service repository for services whose input parameters are satisfied in the P_0 layer. These services are named as available services and added into the A_0 layer. All parameters in the P_0 layer and available services outputs are added to the P_1 layer, so the P_1 layer is a superset of the P_0 layer. The search graph is extended layer by layer and this process ends when the graph reaches a layer having reached the goal state or no more services can be added into the graph. If the goal state can not be found in the search graph, the problem can not be solved. Otherwise, the problem can be solved. Then after the graph search has been successfully constructed, a backward searching algorithm is applied to retrieve solutions from the goal layer to the initial layer. To find all composite plans or find the plan with the best QoS value, we need to check all services combinations. In this case, the complexity of the backward search is NP-complete.

Some AI planning approaches [60] address the web service composition problem

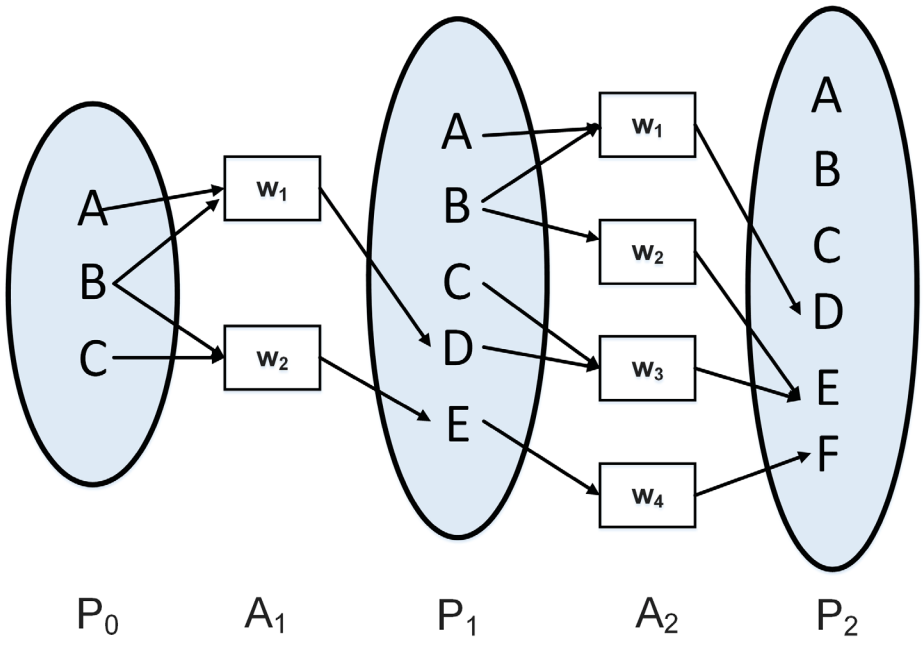


Figure 3: Example of a planning graph.

through the provision of high-level generic procedures and customizing service requesters constraints. Moreover, there are AI planning-based approaches [70, 69] using heuristic algorithms to compute the cost of achieving individual parameters starting from the inputs, and search to approximate the optimal sequence of services that properly connect inputs to outputs. Many of the AI planning approaches support the use of precondition and effects to describe services [79]. For instance, SWORD [78] is a developer toolkit for building composite web services using rule-based plan generation. In SWORD, a service is modeled by its pre-conditions and post-conditions and a web service is represented in the form of a Horn rule that denotes that post-conditions are achieved if the preconditions are true. Graphplan [33], a planning technique which uses initial conditions, goals and information to reduce the number of searches, has been adopted in service composition to find a solution [97, 102]. This solution modifies the standard graph plan algorithm to extract a composition solution that the planning graph construction can build. The time complexity of this process is polynomial in the length of the initial state and the number of services.

Integer Linear programming (ILP) has been applied to find an optimal

solution (which require finding all possible solutions) for the web service composition problem [105, 12, 49]. In this approach, some or all of the variables are restricted to be integers. Berbner et al. propose a mixed integer programming formulation which is more feasible in dynamic real-time environments [12]. Zeng et al. apply dynamic global optimization method in composition process by considering multiple non-functional criteria in service selection process [105]. Such ILP technique has the drawback of exponential computational complexity and cost when the number of variables increases [49]. Taking this into consideration, Alrifai and Risse combine global optimization with local selection techniques to solve this problem [2]. They decompose each QoS constraint into a set of local constraints which serve as upper bounds, then, local selection is applied independently. Their method can find a close to the optimal solution while reducing the computational time. Canfora et al. use Genetic Algorithms to handle QoS attributes which have non-linear aggregation functions [19]. Experimental results show this method can keep a constant timing performance [49].

Considering all above discussed approaches, there might be a situation where no feasible service composition solution is found after a certain time of running service composition algorithm. Lin et al. propose a Relaxable QoS-based Service Selection Algorithm (RQSS) to find an approximate solution [52]. This approach is a relaxation method in which the algorithm relaxes the degree of constraints and recommends a similar solution in case no feasible solution satisfies the constraints. Mabrouk et al. present a near-to-optimal method in which the whole composition task is divided into pieces [104]. In this method, they use K-means to group service candidates into multiple QoS levels and select multiple services for each subtask. The main concern with this method is that it may fall into the local maximum problem [55].

Huge search space and the identification and removal of redundant services restrict web service composition approaches that can generate all possible solutions for a web service composition problem. The above-mentioned approaches are all in memory composition methods which work when data fits in RAM and the searching space is limited by the available physical memory. To fix this problem, researchers have

been motivated to use a database to solve service composition problem. Utkarsh et al. [81] present a Web Service Management System (WSMS) which transforms the service composition problem into a query optimization problem in database. In the WSMS system, they first build virtual tables for services‘ inputs and manage service interfaces, then, they use a multi-threaded pipeline executive mechanism to improve the efficiency in searching services. In [49], Jing et al. propose a novel relational database approach for automated service composition. In this solution, all possible service combinations are generated beforehand and stored in a relational database. Then, the system composes SQL queries to search in the database and return the best QoS solutions based on a user request [49].

In addition, there are approaches that focus on removing redundant services from forward-generated composite plans. Zheng and Yan [107] propose four strategies to prune redundant services in the forward expansion stage of the planning-based approach. They avoid adding a service if its outputs are existing in previous proposition levels or are produced by other existing services, delay adding a service whose outputs are not used in the next action layer, and stop expanding the graph if the goals are found in a proposition layer. Lin et al. [107] propose a service threshold mechanism to reduce the number of services in the search stage, that is, fix the maximum number of services can be invoked in a solution. Also, solutions with with more services but similar to a shorter solution are removed. In the particle swarm optimization composition technique [26], the authors use a greedy optimization algorithm to extract non-redundant solutions from a graph showing all service connections. To check whether or not a service is redundant, the simplest way is to remove that service and recompute the QoS value of the new plan. The redundant service removal method is suitable as a last step to optimize the composition solution [24]. Kwon and Lee [41] propose a two-phase algorithm based non-redundant composition system in which the forward phase finds candidate compositions and the backward phase decomposes compositions into several non-redundant solutions.

2.2 Context and Constraint in Web Service Composition

Context-aware systems have been discussed in many researches [61, 7, 77] and several approaches have been proposed to represent contextual data such as key-value models [13], graph models [39], ontology models [94], and logic models [82, 58, 59] . Although many researches have been done on using context-aware computing in different domains, the notion of a "context" is somewhat vague to define in different areas including web service environment. This is because context information is dependent on individual systems. This is due to the fact that one type of information might be considered as context information in one system but not in another one. In some researches [85, 56] in web service computing area, context is considered as any additional information that can be used to improve the behavior of a service in a specific situation. Therefore, without such additional information, the service should be operable as normal but with context information it is arguable that the service can operate better or more appropriately [85]. In this sense, a context-aware service as a smart web service is defined by Manes as: *"a web service that can understand the situational context and can share that context with other services"* [56].

Some other researches specifically define context-based on sources of contextual data. In web service environment, those sources are service requester, service provider, service users and service execution environment. A Context-aware Service Oriented Architecture (CA-SOA) is an architecture that proposes different components to support service discovery based on the context of service requesters. This approach uses ontologies to model context description linking service requester and web services in service discovery [23]. Akogrimo is a context-aware web service based system which aims at supporting mobile users to access data, knowledge, and computational services on a grid. Akogrimo concentrates on user context that is related to situations of mobile users, such as user presence and location, and environmental information [73]. Han et al. present the Anyserver platform which supports context-awareness in mobile web services [36]. The Anyserver platform utilizes various types of context

information, such as device information, networks, and application type. In [85], Truong et al. represent different components of a context-aware web service-based application (Figure 4). In their view, basic components are divided into two parts. In the top part, there are context-aware applications including context-aware services that communicate based on web services standard protocols. In the bottom part, various supporting components for context- awareness are present. These components are either part of some web services or web services themselves. In addition, in the client-service view, components are distinguished based on the client (service requester) role and service (service provider) role. On the left-hand side, web applications utilize web services. Some supporting components can also provide context information associated with clients, such as sensors. On the right-hand side, various context-aware services offer different services to the clients. These services can interact with each other and will utilize various supporting components to be context-aware.

More specifically, context is also discussed in some researches relate to web service composition. In [61], a novel matching framework for web service composition is proposed. The framework combines the concepts of web service, context, and ontology. The framework relies on an ontology-based categorization of service contexts to match requester and providers context. In [108], Rasch et al. propose a proactive service discovery approach for pervasive environments using the user‘s current context. They consider context as any available data in service environments, which is modeled using an ontology. In [84], a context-aware web service composition framework based on agent modeling. This framework puts context awareness and agent technology into the execution of web service composition, aims to improve the quality of service composition considering service users context.

With respect to all the above-mentioned approaches, our perspective to context-awareness in web service composition is different in important aspects. We do not consider context only as additional information that can be used to improve the behavior of a service in a situation. Rather, we define context as any information relates to service requesters, service providers, service users and the

execution environment of a composite service that can affect the execution of the composite service. However, we acknowldge that a service cannot be executed in all possible contexts. Therefore, different types constraints define restrictions to verify whether or not a service can be executed in a particular context. As we discussed in Chapter 1, constraints are defined by service requesters, service providers and the execution environment of composite services. Most researches in the areas of QoS-aware service compositions and service discovery discuss the role of service requester constraints in service composition [52, 55]. However, there are few approaches that focus on constraints imposed by service providers or by the execution environments of composite services. In [92], a constraint-aware approach for web service composition is proposed. This approach proposes a simple formal expression to describe service constraints that are imposed by service providers. Many of the AI planning approaches [78, 107] support the use of precondition and effects to describe services [79]. For instance, SWORD [78] is a developer toolkit for building composite web services using rule-based plan generation. In SWORD, a service is modeled by its pre-conditions and post-conditions and a web service is represented in the form of a Horn rule that denotes that post-conditions are achieved if the preconditions are true. However, looking through many AI planning approaches, the pre-conditions express the required input parameters, effects, and expected services outputs which could only be used for reasoning during planning. It is clear that this representation of pre- condition and effects cannot express other limitations of services such as service usage constraints as we discussed earlier. In [95], Wu et al. propose a QoS-aware optimal service composition approach which aims to maximize the overall QoS value of the resulting composite service instance while meeting service requester specified global QoS constraints. They propose the concept of Generalized Component Services (GCS), which is defined semantically, to expand the selection scope so as to achieve a better solution compared to other approaches. Bentaleb et al. propose a composition model architecture based on cloud SaaS that takes not only the quality of service, but also cloud computing and context-awareness aspect of the composition into consideration. This approach provides a solution to optimize

the quality of service given to the user by taking the user context into account [9].

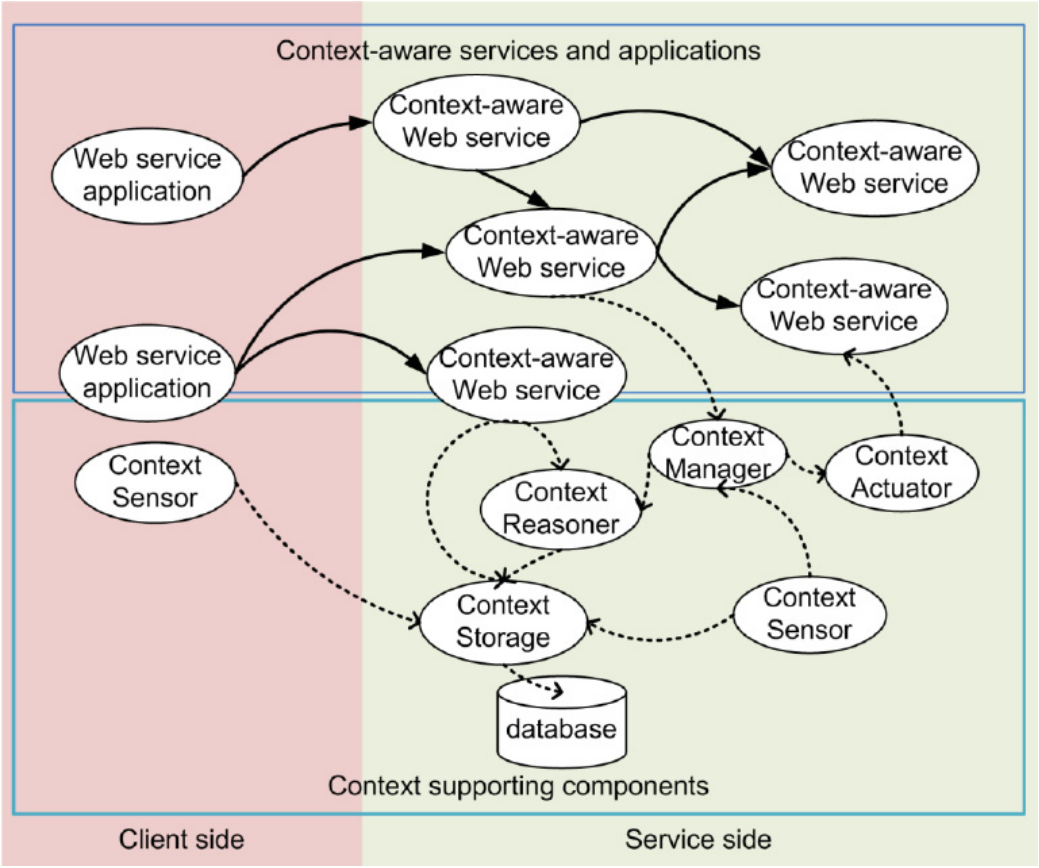


Figure 4: Basic components in a web service-based context-aware system [85].

Considering the problems and objectives we discussed in Chapter 1, there needs to be a context/constraint-aware model and approach which could represent, verify and apply different types of constraint on composite services. Table 2 provides a comparison among some of recent context/constraint-aware web service composition approaches. The table compares different approaches from different perspectives. It shows the type (source) of context and constraint that is supported by each approach. Almost all discussed context-aware approaches consider the context of service requesters. In addition, provider and execution environment related constraints which also need to be considered at execution time of composite services is also discussed in only a few approaches.

Table 2: Current context/constraint-aware web service composition approaches

	Year	Modeling Approach	Context				Constraint			Constraint-aware Execution
			Requester	Provider	Environment	User	Requester	Provider	Environment	
Wang et al. [92]	2014	Graph Search	-/+	+	-	-	-	+	-	-/+
Wang et al. [93]	2015	Graphplan	-/+	+	+	-	-	-/+	-	-/+
Lin et al. [52]	2011	relaxation	+	-	-	-	+	-	-	-
Ponnekanti et al. [78]	2006	AI Planning	+	-/+	-	-	+	-	-	-
Quanwang et al. [95]	2016	AND/OR graph	+	-	-	+	+	-	-/+	-
Yan et al. [98]	2012	AI planning	+	-	-	-	+	-	-	-
Bentaleb et al. [9]	2017	Architecture	+	-	-	+	-/+	-	+	-/+
Medjahed et al. [61]	2007	Framework	+	+	-	-	-	-	-	-
Rasch et al. [108]	2011	Architecture	+	-	+	+	-	-	-	-
Sun et al. [84]	2013	Framework	+	-	-	+	-	-	-	-

2.3 Failure Recovery and Adaptation in Composite Web Services

The execution of composite web services may fail to work properly and recovery is possible but may require to change their structure. This could happen as specific services inside the plan may fail to work properly or emerging requirements and constraints may be imposed on the composite service at runtime. When services inside a composite service fail to work properly, the process to get the composite service back to work is called *failure recovery*. Besides, there might be situations that new requirements or constraints imposed on a composite service at runtime. The process to adapt a composite service to newly added constraints and requirements that are imposed by service requester or execution environment of the composite service is called *composite service adaptation*.

Failure of a composite service could happen as a result of different reasons such as unavailability of services inside the plan during the execution of the composite service, or failure in verification of different types of constraint. Unavailability of a service could have many reasons including SLA violations, technical issues, etc. Explained in general terms, constraint verification failure happens when a service cannot be executed in a certain context. In this situation, constraints verification, which check whether the service can be executed in a given context, fails the execution at runtime. Therefore, in both cases, there needs to be a recovery mechanism to recover the broken plan in order to continue to provide services to the users. It should be noted that there is a difference between failure recovery resulting from the unavailability of component services and failure resulting from service constraint verification. When a service inside a component service is not available, it will be excluded from a composite service plan by the recovery process and any plan using this specific service will no longer be valid. However, when service constraint verification fails in the execution of the composite service, the recovery process should find an alternative plan to complete the execution. This should also take into consideration that the failed service may still be valid for some other executions in a different context, hence it

is not required to be excluded from all available plans. Going through the recovery process for every constraint verification failure inside the composite service could add considerable overhead to system performance during the execution of the composite service. In addition, new constraints and requirements from service requester and execution environment of the composite service might be imposed, therefore changing the execution context and make it invalid for the composite service to be executed in the new context. Therefore, the composite plan should be adapted in a way to meet emerging requirements needs and change the composite service structure based on the new constraints. These problems are addressed in different domains including *web service composition adaptation* [42], *failure recovery* in web service composition [57, 99] and *web service composition transactions* [32, 29, 27, 96].

Re-planning is the simplest of all approaches when a few services fail to work. In this approach, the composition mechanism starts making a new composite service avoiding failed services. Many re-planning based approaches have been proposed for recovery and adaptation of composite services [16, 20]. *Replacement* is another approach to react to a faulty service [34, 21]. Replacement is limited to 1-1 substitution and it focuses on finding a replacement for a broken service by another one. There are different solutions for this, such as finding a service that can use less input and produce more outputs than the original one. Replacement is preferred when a service is faulty or has a bad QoS. However, replacement cannot deal with the needs of adding or removing services in a composite service plan. In [103], Yu and Lin propose a Composite Service Process Reconfiguration (CSPR) algorithm which uses backup services to reconstruct the process. The backup path is produced offline during the service composition. Unfortunately, their backup method is feasible when there is only one faulty service. The algorithm fails to find a replacement if two or more services fail. Replacement is an efficient solution in terms of computation time. However, the limit of replacement is that a broken service often cannot be replaced by another unique service.

Re-composition and *repair* are two approaches that support 1-n substitution. Re-composition re-builds the broken service by a 1-n substitution. Re-composition

could also go further by using an entirely different set of services and hence would correspond to an n-m substitution and applied in [53, 54, 106]. Lin et al. [53] present a reconfiguration solution to support multiple faulty services where an algorithm is designed to recognize reconfiguration regions. This algorithm stops when a solution is found or the number of services exceeds expectation. Later, they extend the work of [53] and implement the solution in the Llama architecture [54]. They claim that the region-based algorithm reduces the re-composition complexity. However, re-composition is time-consuming and quite costly since a new business process should be computed [49]. Zhai et al. [106] propose a services reconfiguration solution to handle multiple service failures in the business process.

Repair is also another solution that goes beyond the limits of service replacement while avoiding re-composition. The term plan repair was first introduced from a theoretical perspective in the AI area [86]. This technique aims not only at keeping most of the above mentioned models as-is (i.e., not recompute them), but also takes benefit from them while computing a corrected composition. As such, repair is a form of heuristic and guided partial re-composition. In case of 1-1 substitution, repair performs as the replacement and is as efficient. In other cases and for added needs, repair yields better computation time than re-composition while retrieving solutions of the same quality [99]. Many of the existing solutions to improve reliability of composite service through improving the flexibility and adaptability of composite service are static. Static approaches are more focused on the idea of adaptation by substitution of the composite service with an alternative composite service [57]. In [22] a framework (A-WSCE) is presented to adapt by defining multiple work-flows and switching among available solutions to keep the composition work-flow available. In addition, there are adaptation solutions for implementing variability constructs at the language level [8, 25, 40] and using model-driven approaches [18, 62]. Language-based approaches use many mechanisms to implement and manage dynamic adaptations at the language level. In [8] monitoring directives are expressed in a web service constraint language, and recovery strategies, which follow the Event-Condition-Action (ECA) paradigm, are stated in the Web Service Recovery Language [1]. Even

some approaches proposed Aspect-Oriented Programming for self-adaptive service compositions [80]. However, it is argued that implementing and managing dynamic adaptations at the language level can become complex, time-consuming, and requires low level implementation mechanisms [1]. Model-driven approaches use models at run-time to support dynamic adaptation of service compositions [18, 62]. However, these approaches are too abstract and their implementation solutions are not clear [1].

AI-planning techniques have successfully been used to support underspecified composition requirements [76, 28, 100]. These planning approaches mainly deal with the problem of adaptability with repair and re-composition [64]. Compared to simple 1-1 substitution, re-composition approaches can do 1-n substitution to adapt a composition [107]. In [99], Yan et al. proposed web service composition repair as an alternative adaptation mechanism to recomposition. This is when repair does as good as replacement when 1-1 substitution is possible, but goes beyond this limit, supporting 1-n or n-m substitution and added needs.

Dynamic transaction support for web services is another approach to ensure that the composite service is executed correctly and achieves the overall desired result [32]. Transactions are an approach employed to address system reliability and fault-tolerance [29] and the goal of service composition based on transactional properties is to ensure a reliable execution of the composite service. Traditional web services transaction processing mechanisms handle an exception by forward and backward recovery approaches [29, 27]. Backward recovery is essentially a form of rollback that unrolls the transaction and restores the original state of the system. Forward recovery approaches attempt to reach the original goal of the composite service by retrying or replacing components and continuing the process [63, 100]. In [96] a framework to optimize the success rate of transactional composite services is proposed. The framework considers the success rate of a service to include it as a candidate in the composition process. In this way, they improve the success rate of composite services completing successfully and thus reduce the need to employ failure recovery approaches.

Wang et al. [88] propose a context-aware architecture for self-adaption in web

service composition. In their perspective, service context describes the properties of the service and the required execution environment of a service. These properties and preferences for services are written by a service provider and updated by user ratings. User context describes requirements and the environment that the service consumer can provide. Device context describes the real execution environment, including hardware and software environment. This architecture contains a context module that is responsible for adapting WSC to the changing at QoS and satisfies the service requesters‘ constraints. In this approach, re- composition in web services is made in a case input and output parameters of the composition problem are changed. In addition, changing the context in this approach is handled according to user-defined personalized policies. Wang et al. [91] present self-adaptive service composition framework based on Reinforcement Learning. This framework uses Markov Decision Process to model web service composition and adapts to the dynamic evolution of service requesters‘ requirements. In their approach, the concrete workflows and services selection is specified based on the environment and the status of services [89, 90].

Table 3 compares some of the above-mentioned failure recovery and adaptation approaches from two main aspects: failure recovery of composite service and composite service adaptation to new requirements and constraints which might be imposed by service providers, service requesters and execution environment of composite services. As we stated before, current failure recovery approaches focus on the problem of unavailability of services in the composite plan. To the best of our knowledge, almost all failure recovery approaches concentrate on failure recovery of composite services in cases of unavailability of sub services. In addition, most approaches only work on the adaptation of composite services to new requirements and constraints imposed by service requesters. In addition, adaptation to constraints of the execution environment is only discussed in few approaches.

Table 3: Current runtime adaptation and recovery approaches

	Year	Techniques	Failure Recovery		Adaptation	
			Unavailability	Constraint Failure	Requester	Environment
Yu et al. [103]	2005	Replacement	+	-	-	-
Yan et al. [99]	2010	Repair/Forward	+	-	+/-	-
Boella et al. [16]	2002	Re-planning/ Backward	+	-	-	-
Lin et al. [53]	2009	Recomposition/Forward	+	-	-	-
Chafle et al. [22]	2007	Re-planning/ Backward	-	-	+	-
Cavallaro et al. [21]	2009	Replacement	-	-	+	-
H. Wang et al. [89, 90]	2016	Q-Learning/Forward	+/-	-	+	-
B. Wang et al. [88]	2014	Re-composition	+	-	+	-
Van Der Krogt et al. [86]	2005	Repair/Forward	+	-	-	-
Jiuyun et al. [96]	2016	Framework	+	-	+	-/+

Chapter 3

Constraint-aware Web Service Composition

Most methods that have been proposed to solve the problem of web service composition only consider input and output parameters of services to solve the composition problem. However, there are other factors that affect composition and execution of composite services such as composite service execution context and constraints. Constraints can be used to express customer requirements on services features. Additionally, most real-world web services have constraints that specify their limitations and usage restrictions. Constraint verification has a significant impact on composition and execution of composite services. In particular, runtime verification of service constraints can result in the failure of the execution of composite services and eventually waste computational resources. Such failures can not always be predicted as the verification of some services depends on execution effects of other services inside a composite plan.

Most of the existing AWSC approaches do not fully consider the problem that component services of a composite service may have individual constraints that need to be verified as part of the composition and execution of the composite service. It is also not possible for them to generate constraint-aware composite services that could be executed considering service constraints, and to eventually minimize service rollbacks resulting from the violation of service constraints at runtime. In a few

existing approaches [92, 93], constraints are embedded inside a composite plan to be verified as the composite service is being executed. However, they are not providing a solution to minimize potential service usage rollbacks resulting from failure and/or recovery.

In this chapter, we first propose the definition of a composite service model including different concepts such as service, constraints and web service composition problem. Second, we propose a novel constraint-aware web service composition approach based on graph plan approach. Then, a novel approach to verify constraints of constraint-aware composite services based on the context of composite service at execution time is provided. Finally, in the analysis and evaluation section we provide mathematical and experimental evaluations to show the effectiveness of our approach compared to other approaches. Our mathematical evaluation discusses the effectiveness of our approach in reducing the number of service rollbacks that are prevented during the failed execution. Then, we implement our novel constraint-aware web service composition approach and use a publicly available dataset generator to evaluate our approach in practice.

3.1 Motivation Scenario and Problem Analysis

Consider a shopping application that consists of a set of tasks: searching for products, submitting an order, paying for the order, and shipping/delivery of the order. A service requester makes a request to the composition engine for a composite service that lets a user order a product ($ProductName$) and ship ($ShipmentConfirmation$) it to his address ($DeliveryAddress$). The service requester also specifies a set of constraints on the composite service. For example, service requester can set a constraint on the total value of the composites service (e.g.the total value must not be more than 10 ($Cost < 10$)). Service cost is the amount of money paid to the service provider to use the service and for a composite service it is the sum of all the sub-services' costs. The available individual services are depicted in Table 4. In addition, the three shipping services have different applicable constraints, e.g., the

standard shipping service (W_3) is available only for products whose $ProductAddress$ and $DeliveryAddress$ is located in Montreal; two-day delivery (W_4) is available only for orders whose $ProductAddress$ and $DeliveryAddress$ is located in the province of Quebec; while another shipping service (W_7) is available for orders in Canada. Given the service requester constraints ($Cost < 10$) and all required input($ProductName$, $DeliveryAddress$) and output ($ShipmentConfirmation$) parameters, Figure 5 shows all possible composition plans that could fulfill the request from the service requester.

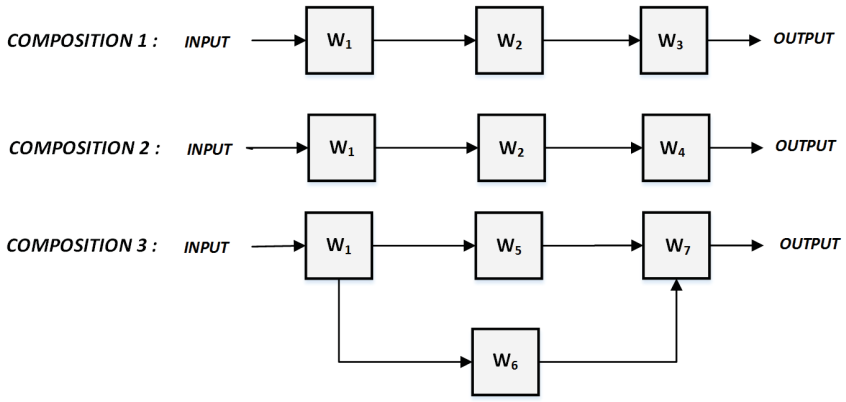


Figure 5: Possible composition plans.

There are three composition plans for the shopping service request that can accomplish the shopping task. In each plan, the shipment service has its constraints based on the $ProductAddress$ and $DeliveryAddress$ of the orders. The set of all parameters whose values are used or modified during execution of a service is called the *data model* (DM) of the service. For example, DM_{W_1} is the data model of W_1.

$$DM_{W_1} = \{ProductName, ProductNumber, DeliveryAddress, ProductAddress\}$$

For a composite service, the data model is the union of the data model of all sub services. During the execution of a composite service, values assigned to parameters of the data model (data model of the composite service) are modified by execution of services inside the composite service. This set of parameters and their assigned values

Table 4: Available services

#	Service	Input	Output	Cost	Constraints
W_1	Search	{ProductName, DeliveryAddress}	{ProductNumber, ProductAddress, PaymentAmount}	0.4	$c_1 = \emptyset$
W_2	Order/Payment	{ProductNumber, PaymentAmount}	{PaymentNumber, OrderNumber}	4	$c_2 = \emptyset$
W_3	Shipment	{PaymentNumber, DeliveryAddress, ProductAddress, OrderNumber}	{ShipmentConfirm}	2	c_{31} = DeliveryAddress $\in$ Montreal c_{32} = ProductAddress $\in$ Montreal
W_4	Shipment	{PaymentNumber, DeliveryAddress, ProductAddress, OrderNumber}	{ShipmentConfirm}	2.5	c_{41} = DeliveryAddress $\in$ Quebec c_{42} = ProductAddress $\in$ Quebec
W_5	Order	{ProductNumber}	{OrderNumber}	3	$c_5 = \emptyset$
W_6	Payment	{ProductNumber, PaymentAmount}	{PaymentConfirm}	3	$c_6 = \emptyset$
W_7	Shipment	{PaymentConfirm, DeliveryAddress, ProductAddress, OrderNumber}	{ShipmentConfirm}	1	c_{71} = DeliveryAddress $\in$ Canada c_{72} = ProductAddress $\in$ Canada

is called the *context* of the composite service. For each execution of the composite service, the context keeps track of execution results and effects of the services inside the composite service and passes the accumulated information further down to the upcoming services in the composite service.

Consider that a user of the shopping service wants to use the composite service to buy a book. Given that the service requester specified cost optimization as a constraint, *composition* 1 (Figure 5) is picked for shopping as the best composite service. At execution time of composition one, after searching for the book (executing W_1), it may turn out that the $ProductAddress$ (i.e. the value assigned to $ProductAddress$ in the composite service context) is Toronto. In this case, after ordering the product and making the payment (executing W_2), the execution of the shipment service (W_3) fails, as the $ProductAddress$ is not in Montreal. In this situation, W_1 and W_2 have already been executed and their executions results need to be rolled back.

It is clear that if the constraints of W_3 were verified right after execution of W_1 (i.e. when the value of $ProductAddress$ becomes known), the execution failure could be predicted (i.e. verified sooner) and less rollback would be required. This example shows that there are constraints (e.g. shipment constraint related to the delivery address) that can be verified before execution of the first service in the composition plan. In addition, even if the parameters can match the respective input/output interface, the service may still be unable to execute correctly if its service constraints are not satisfied. Some service constraints can only be verified during the execution (e.g. constraint related to $ProductAddress$) and their failure can fail the execution of the whole composite service. These sorts of failures cannot be avoided, as the verification depends on the values that are going to be produced during the execution of the composite service (e.g. $ProductAddress$). Having a constraint-aware composite service which is aware of its component services' constraints during the execution can help to catch upcoming failures earlier and thus avoid some of the incurred rollbacks. In this way, for example, if the composite service execution system is aware of component services' constraints during the execution process, it

could check the shipment service constraint right after execution of the search service and prevent the execution of the payment service when the product address is not in the Montreal area. Therefore, the issue is to design constraint-aware plans which enable more effective constraint verification by catching upcoming failures as soon as possible inside a composition plan.

3.2 Chapter Methodology

We are motivated to keep track of the context and the constraints, and verify all related constraints as soon as the context changes. In this way, the upcoming failure can be caught sooner and the number of rollback penalties could be minimized. To do that, we formally express dependencies among services. Using these dependencies, we plan to move back verification points of service constraints inside a constraint-aware composite service. To be able to manage verification points of constraints inside composite services, we propose a novel composite service model and graph plan-based algorithms to generate constraint-aware composite services. We are interested to use graph plan-based approach which is a powerful approach to generate possible constraint-aware composition plans for a service composition request. In addition, as graph plan generates composite services in different service layers it makes it easier to adjust service constraints in different service layers of composite services. This chapter has two different contributions. First, we propose a novel constraint- aware service model and algorithms to generate constraint-aware composite service plans. Second, we propose an efficient constraint verification method to minimize the cost of constraint verification failure by minimizing wasted service executions. To evaluate our proposed approach, mathematical and experimental (based on a publicly available dataset) evaluations are provided. Mathematical evaluation proves the effectiveness of our approach compared to other constraint verification approaches regarding the number wasted service executions that are saved. For experimental evaluation, we implement our proposed algorithms to generate constraint-aware composite services. Then, we use a publicly available dataset generator to show the effectiveness of our

approach compared to current constraint verification approaches.

3.3 Composite Service Model

To have a clear understanding of concepts and problems, at first we define a formal model including different concepts for web service composition.

Definition 1. *A **Service** is defined as a tuple* $s =< I, O, C, E, QoS >$ *where:*

- ***I*** *is a set of ontology types representing the input parameters of the service.*
- ***O*** *is a set of ontology types representing the output parameters of the service.*
- ***C*** *is a set of constraint expressions representing limitations on service features.*
- ***E*** *is a set of ontology types representing parameters whose value are affected as a result of the execution of the service.*
- ***QoS*** *is the set of quality parameters of the service.*

In our definitions, we used ontology to define concepts (ontology type) and the relations between them. QoS criteria determine usability and utility of a service [74]. Besides the service definition, we need to have a definition for *service data model*.

Definition 2. *A **Service Data Model** is a set of ontology types representing all the parameters a service accepts as input, produces as output and modifies during its execution.*

The data model of a composite service is the union of the data model of all its component services' data model. For example, Figure 6 depicts the composite service for services discussed in Table 5. The composite service gets $\{a, b\}$ as inputs and produces $\{f\}$ as output parameter. The data model of the composite service (D_{w_c}) is $\bigcup D_{w_i} = \{a, b, c, d, e, f\}$.

We also need to define *Constraint* to specify the limitations on service features (input/output and QoS) that must be considered to ensure correct execution

Table 5: Service specifications.

Service	Input	Output	Data Model
W_1	a	c	$D_{w_1} = \{a,c\}$
W_2	b	d	$D_{w_2} = \{b,d\}$
W_3	c	e	$D_{w_3} = \{c,e\}$
W_4	d,e	f	$D_{w_4} = \{d,e,f\}$

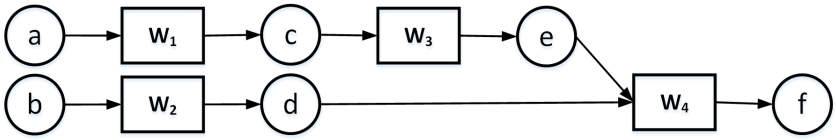


Figure 6: A sample composite service.

of services. A constraint is a function that maps a service feature to a set of values. To express constraints formally, we use the following definitions:

Definition 3. *A **Constraint** is an expression that can be evaluated to either true or false. For simplicity, we restrict ourselves to expressions of the form:*
< feature >< operator >< literalValue >, where:

- $<$ *feature* $>$ *represents an input, output or quality parameter of a service which is an ontology type.*
- $<$ *operator* $>$ *represents operators such as* $=, \neq, <, >, \leq, \geq, \in, \subset, \supset, \subseteq, \supseteq$.
- $<$ *literalValue* $>$ *represents a value or a set of values of the same data type as the expression feature.*

For example, $c = cost \leq 10$ expresses a constraint on the *cost* (QoS feature) of a service. In addition, there needs to be a mechanism to evaluate constraint satisfaction. Therefore, we define **Satisfaction Degree** as a mechanism to verify constraints.

Definition 4. *If c is a constraint on feature f and v is a value assigned to feature f, **Satisfaction Degree** ($SD(c, v)$) is a function that calculates a quantitative measure to evaluate the satisfaction of c according to v.*

For example, if $c = cost \leq 10$, for any value v assigned to cost ($cost \leftarrow v$) :

$$SD(c, v) = \begin{cases} true & v \leq 10 \\ false & \text{otherwise} \end{cases}$$

In addition, when there are more than one applicable constraint, **General Satisfaction Degree** verifies satisfaction of all constraints.

Definition 5. *If C is a set of constraints on a service feature f and v is a value assigned to feature f, **General Satisfaction Degree** ($GSD(C, v)$) is a function that calculates a quantitative measure to evaluate the satisfaction of the value assigned to f according to all constraints in C.*

If C is a set of constraints that includes n constraints that are applicable to f then $GSD(C, v) = \prod^n SD(c_i, v)$. For example, $c_1 = payment\_Method \notin \{Visa\}$ and $c_2 = payment\_Method \in \{Visa, MasterCard\}$ are two constraints expressing limitations on $payment\_Method$. Then, the combined constraint representing both restrictions is defined as $c_T = payment\_Method \in \{MasterCard\}$. In addition, for $C = \{c_1, c_2\}$:

$$GSD(C, v) = \begin{cases} true & v \in c_T \\ false & \text{otherwise} \end{cases}$$

To start the composition process, a composition request is made according to specifications given by a service requester. We define a web service composition request as:

Definition 6. *A **Service Composition Request** R is a tuple $R =< I, O, QoS, C >$ where:*

- ***I** is the set of ontology types representing the input the customer can provide.*
- ***O** is the set of ontology types representing the output expected by the customer.*
- ***QoS** is the set of quality parameters expected from the service by the customer.*

- ***C*** *is the set of constraints representing limitations of service requester.*

Current AWSC approaches design composite service *solution plans*, which is a workflow of web services, to accomplish the task expressed by the service composition request. We define a constraint-aware plan to accomplish a service composition request as:

Definition 7. *A* ***Constraint-Aware Plan*** *is a directed graph extracted from the search graph in which each node is a service-node* $< C_S, service >$*, using initial parameters (R.I), whose successive application of services of nodes is eventually generating the goal parameters (R.O).*

For each service-node ($< C_S, service >$) in a constraint-aware plan, C_S refers to the set of all service constraints that are required to be verified before execution of *service* inside the plan, which is initiated to C_S by default. In this paper, by execution of a service-node we mean executing the *service* of the service-node and by verification of the node we mean verification of all constraints of the service-node (C_S). We also define sets of predecessors and successors of a service-node as follow:

Definition 8. *The* ***predecessor*** *set of a service-node in a constraint-aware plan represents the set of all services-nodes that must be executed before execution of the service-node, and the* ***successor*** *set represent the set of all services-nodes that are going to be executed after the execution of service-node in the constraint-aware plan.*

For example, for W_7 in *composition 3* (Figure 5), the predecessor and successor sets are: $predecessors(w_7) = \{w_5, w_6\}$ and $successors(w_7) = \emptyset$.

3.4 Constraint-Aware Service Composition

This section discusses our novel planning-based service composition approach to construct constraint-aware composite plans based on a composite service request. Then, we propose our novel constraint verification approach inside a composite service.

3.4.1 Service Composition

Algorithm 1 discusses our planning-based service composition approach. This approach includes two stages: (1) a forward expansion stage (Algorithm 2) that constructs a *search graph* from a service composition request (line 2) and (2) a backward searching stage (Algorithm 4) that retrieves *solution plans* from the search graph (line 5). Finally, it generates constraint-aware plans (*cnstr_plans*) and adjusts constraint verification points inside them (lines 12-21) based on the methods we will discuss in Section 3.4.2. The search graph is a graph that is the result of forward expansion and includes at least one possible solution to the service composition request. In the search graph, each service belongs to a service layer inside the graph. Each successive layer represents one step further away from the initial service of the search graph.

In Algorithm 2, the composite request's initial parameters ($R.I$) are added to the initially empty set of parameters produced ($prdSet$) by each successive layer of the graph. Then, it searches the service repository for services whose input parameters are satisfied by the parameters in $prdSet$. Adding new services also require adding some values to the search graph by at least adding a new parameter (line 2-4 of Algorithm 2). These services are available services who are then added to the next service layer, provided that they do not violate any of the composition request constraints ($R.C$) (line 6). Then, all parameters in the selected services‘ outputs are added into the set of parameters produced by the search graph ($prdSet$). In this forward expansion mechanism, the search graph is extended and services are added layer by layer. This process ends when there are no more services in the service repository that can be added to the search graph. If some element of $R.O$ cannot be included in $prdSet$, the problem can not be solved and results in failure, otherwise, the problem can be solved and the search graph is returned (lines 13-15).

In our approach, services can be composed in sequence or in parallel. *AddService* (Algorithm 3) decides the order of insertion of the service (*newService*) in the search graph by identifying its set of predecessor services. Starting from services in

Algorithm 1 Service Composition

Input: R (composition request), SR (set of available services).
Output: *plans* (a set of constraint-aware plans, or failure).

1: $serviceSet = \emptyset; plans = \emptyset$
2: $searchGraph = ForwardExpansion(R, SR)$
3: **repeat**
4: l = maximum layer index in the search graph
5: $ServiceSet$ = all services in layer l of the serach graph
6: $serviceSet = BackwardSearch(searchGraph, ServiceSet, \emptyset, l)$
7: $plan = constructPlan(serviceSet)$
8: **if** $(plan \notin plans)$ **then**
9: $plans = plans \cup plan$
10: **end if**
11: **until** $(No\ more\ plan\ can\ be\ added\ to\ the\ plans)$
12: **if** $(plans \neq \emptyset)$ **then**
13: **for** (each $plan \in plans$) **do**
14: **for** (each $service \in plan$) **do**
15: $serviceNode.service = service$
16: $serviceNode.C_s = service.C$
17: $cnstrAwarePlan = cnstrAwarePlan \cup serviceNode$
18: **end for**
19: $cnstrAwarePlan = adjustConstraint(cnstrAwarePlan)$
20: $cnstr\_plans = cnstr\_plans \cup cnstrAwarePlan$
21: **end for**
22: **return** $cnstr\_plans$
23: **else**
24: **return** $failure$
25: **end if**

Algorithm 2 ForwardExpansion

Input: R (composition request), SR (set of available services)
Output: *searchGraph* (search graph generated by forward expansion).

```
1: searchGraph = null; prdSet = R.I
2: repeat
3:    for each service in SR do
4:       if (service.I ⊆ prdSet) and (service.O − prdSet ≠ ∅) then
5:          l = AddService(searchGraph, service)
6:          if (CheckConstraints(l, R.C)) then
7:             searchGraph = l
8:             prdSet = prdSet ∪ service.O
9:          end if
10:      end if
11:   end for
12: until (No service could be added to the search graph)
13: if (R.O ⊂ prdSet) then
14:    return searchGraph
15: end if
16: return failure
```

the first layer, Algorithm 3 searches into the search graph, finds services that produce some of the input parameters of the new service and adds them to the predecessor set of the new service (lines 4-8).

To retrieve a solution plan (*plan*) from the search graph, Algorithm 4 recursively extracts a sequence of service sets using a backward-chaining strategy, which can reach the goal parameters ($R.O$) from the initial parameters ($R.I$). Each time the algorithm backtracks, it chooses a subset of services (*serviceSet*) from the power set of predecessor services (*preSrvSet*) of selected services in the last recursion (line 20-24). If the backtracking gets to the first layer of the search graph, it checks to make sure the input set of selected services of the first layer are available in the initial parameters set (line 7-9). In addition, if the output set also includes output parameters, *planSet* is returned as a solution plan. Finally, the function *ConstructCompositionPlan* (Algorithm 1, line 6) discards all the unnecessary services in *planSet* to minimize the number of component services in the final solution plan and then arranges these service sets in sequence.

Algorithm 3 AddService

Input: *searchGraph* (a search graph), *newService* (A new service)
Output: *searchGraph* (a search graph includes *newService*)

1: $layer = 0; newIn = newService.I$
2: **while** ($layer \leq$ maximum layer index in *searchGraph*) **do**
3: $serviceLayerSet =$ all services in layer *layer* of searchGraph
4: **for** (each $service \in serviceLayerSet$) **do**
5: $prdSet = service.O \cap newIn$
6: **if** ($prdSet \neq \emptyset$) **then**
7: $newService.predecessor = newService.predecessor \cup service$
8: $newIn = newIn - prdSet$
9: **else**
10: **return** *searchGraph*
11: **end if**
12: **end for**
13: $layer = layer + 1$
14: **end while**

3.4.2 Constraint Verification Management in Web Service Composition

As we discussed in Section 3.1, component services in a constraint-aware plan have constraints that must be verified during the execution. It means each service constraint needs to be verified before its execution inside a constraint-aware plan. Figure 7 shows how constraints of a component service are verified during the execution of *composition 1* which is a constraint-aware composite service. Each service-node ($< C_S, service >$) in a constraint-aware composite service is represented as a sequence of two symbols: the constraint represented by a yellow diamond (C_S) followed by the service represented by a gray square (W).

For any Input parameters ($ProductName$, $DeliveryAddress$) *composition 1* executes W_1 and W_2 and verifies shipment constraints (C_{31} and C_{32}) before the execution of W_3. During the execution of *composition 1*, if the $ProductAddress$ address of a product is not in Montreal, verification of C_{32} fails the execution of *composition 1* and execution results of W_1 and W_2 must be rolled back.

However, this is not the optimal way to verify constraints of component services in *composition 1*, as $DeliveryAddress$ is known from the beginning of the execution.

Algorithm 4 BackwardSearch

Input: *searchGraph* (a search graph on which the BackwardSearch is applied), *preSrvSet* (set of predecessor services), *planSet* (the set of services in the solution plan), l (the layer number from which to start the search)

Output: *planSet* or failure

1: $S =$ all services in layer l of the search graph
2: $serviceSet = preSrvSet \cap S$
3: $planPowerSet = powerSet(S)$
4: **for** (each $set \in planPowerSet$) **do**
5: **if** $((serviceSet \cap set) = \emptyset)$ **then**
6: Continue
7: **end if**
8: $planSet = planSet \cup set$
9: **if** $(l = 1)$ **then**
10: **for** (each $service \in set$) **do**
11: $inputSet = inputSet \cup service.I$
12: **end for**
13: **if** $(inputSet \not\subset R.I)$ **then**
14: $Continue$
15: **end if**
16: **for** (each $service \in planSet$) **do**
17: $outputSet = outputSet \cup service.O$
18: **end for**
19: **if** $(R.O \subset outputSet)$ **then**
20: **return** $planSet$
21: **end if**
22: **else**
23: **for** (each $service \in set$) **do**
24: $preSet = preSet \cup service.predecessors$
25: **end for**
26: **if** $(preSet \neq \emptyset)$ **then**
27: **return** $BackwardSearch(searchGraph, preSrvSet \cup preSet, planSet, l - 1)$
28: **else**
29: **return** $Failure$
30: **end if**
31: **end if**
32: **end for**

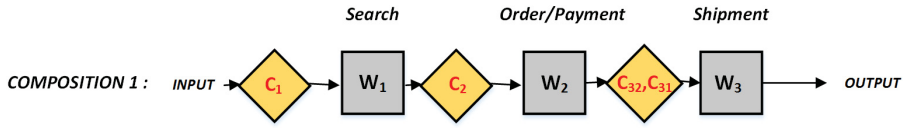


Figure 7: Context-aware composite service

As such, unnecessary execution of W_1 and W_2 could be avoided by verifying C_{31} before the execution of W_1. Besides, the verification of C_{32} depends on the execution result ($ProductAddress$) of W_1. Therefore, if C_{32} were to be verified after execution of W_1, the execution system could avoid unnecessary execution of W_2. In our approach, as the constraint-aware plan is being constructed, the optimal point to verify individual service constraints during the execution is calculated. The optimal way to verify constraints in *composition 1* is depicted in Figure 8.

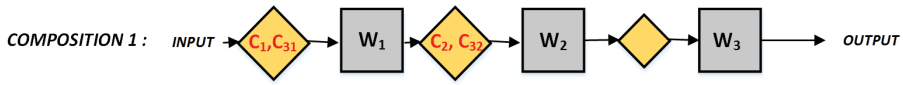


Figure 8: Context-aware composite service with adjusted constraints.

To do that, for C_{32} in *composition 1*, the composition algorithm needs to find the last service in the plan which changes the value of $ProductAddress$ (which is W_1). Then, the algorithm could verify C_{32} right after the execution of W_1. In Definition 1, E is defined as a set of parameters whose assigned values are changed as a result of the execution of the service. Therefore, when the composition algorithm adds W_3 to the constructed plan of W_1 and W_2, it looks back through the plan and finds the last services which have $DeliveryAddress$ (feature related to C_{31}) and $ProductAddress$ (feature related to C_{32}) in their set of changed parameters (E). Constraints like C_{31}, whose feature value does not change before W_3, are added to the beginning of the plan to be checked before execution of any component service in the plan [45].

Algorithm 5 implements our constraint verification approach. Starting from the second layer of the constraint-aware plan, the algorithm moves the verification point of every constraint of each service in this layer back into the plan (lines 9-28). To do that, a service-node ($preNode$) from service-nodes in previous layers of the constraint-aware plan, which also belongs to the predecessors set ($preSet$), is picked. Then, the constraint is moved to be verified before the execution of all successor service-nodes, if the service-node ($preNode$) could affect the value of the feature to which the constraint applies (lines 15-19). This process is repeated until the constraint is moved back to the most efficient verification point. In the case where there is no

Algorithm 5 adjustConstraint

Input: *constraintPlan* (a constraint-aware plan)
Output: *constraintPlan* (a constraint-aware plan with adjusted constraints)

```
1: l_1 = 2
2: while (l_1 ≤ maximum layer index in constraintPlan) do
3:    layerSet = all service-nodes in layer l_1 of constraintPlan
4:    for (each serviceNode ∈ layerSet) do
5:       preSet = serviceNode.predecessors
6:       constraintSet = serviceNode.service.C
7:       for (each constraint ∈ constraintSet) do
8:          repeat
9:             if (preSet = ∅) then
10:               Add the constraint to the beginning of the constraintPlan
11:               break
12:            end if
13:            preNode = a node of preSet with the highest layer
14:            if ( constraint.feature ∈ preNode.service.E) then
15:               for (each sNode ∈ preNode.successors) do
16:                  sNode.C_s = sNode.C_s ∪ constraint
17:               end for
18:               constraintSet = constraintSet − constraint
19:               break
20:            else
21:               preSet = preSet − preNode
22:               preSet = preSet ∪ preNode.predecessors
23:            end if
24:         until (preSet ≠ ∅)
25:      end for
26:   end for
27:   l_1 = l_1 + 1
28: end while
29: return constraintPlan
```

preceding service-node affecting this value, the constraint is moved to the beginning of the constraint-aware plan (lines 11-13).

Finally, it should be noted that a constraint-aware plan is executed differently compared to the regular composite services. Starting from the first service-node (e.g. $< C_1, S_1 >$), in each step all service constraints of current service-node are verified based on the execution context of the composite service. Then, if the verification of none of the service constraints in C_1 fail, the service (S_1) is executed and modifies the composite service context. This process continues until all services in the composite service have been executed.

3.5 Analysis and Experimental Results

This section includes analysis and experimental results of the constraint verification approach discussed in Section 3.4.2. We discuss the effectiveness of our approach in reducing the number of service rollbacks that are prevented during the failed execution of composite services as a result of constraint verification. Then we compare our approach with two other approaches A and B. Approach A discusses the situation where there is no constraint verification during execution of composite services and approach B is similar to current constraint-aware approaches that verify constraints of each concrete service before calling it inside the plan.

Approach C is our proposed constraint-aware verification method that was discussed in Section 3.4.2. The goal of our analysis is to show what percentage (on average) of service rollbacks are prevented using our approach C compared to other approaches (A and B). In each analysis, we consider the worst case scenario for the composition of n different services when all services are composed in sequence. The following lemmas analyze the number of service rollbacks required for each of the approaches. In each lemma, we assume that the probability of failure during the verification of any service constraint in a plan is the same.

In addition, n different services are composed in different ways (parallel or sequence) in a composite plan. The maximum number of rollbacks are imposed

to the system when all services are composed in sequence (worst case). Approach A represents the situation where there is no constraint verification during execution of composite services.

Lemma 1. *Let $P = \{w_1, w_2, .., w_m\}$ be a solution plan of m services. Let $C = \{c_1, c_2, .., c_m\}$ be a set of constraints defined over P. In average, the number of required service rollbacks using approach A is $m/2$.*

Proof. For plan P, the number of required service rollbacks can vary from 0 (the best case where the plan can be executed completely) to m (when verification of the last service in the plan fails and result into m service rollbacks). Therefore, if a service is in position i in the plan, its failure could result in i service rollbacks. The average number of service rollbacks T_{avg} is thus:

$$T_{avg} = \frac{0 + 1 + 2 + ... + m}{m + 1} = \frac{m(m + 1)/2}{m + 1} = m/2$$

□

Approach B is similar to what is used in current constraint verification approaches that verify constraints of each component service directly before its execution.

Lemma 2. *Let $P = \{w_1, w_2, .., w_m\}$ be a solution plan of m services. Let $C = \{c_1, c_2, .., c_m\}$ be a set of constraints defined over P. In average, the number of service rollbacks using approach B is $1/2(\frac{m(m-1)}{m+1})$.*

Proof. In approach B, the constraints are checked before execution of services. The number of service rollbacks can vary from 0 (the best case where the plan can be executed without any problem) to $m - 1$. Therefore, if a service is in position i of the plan, its failure could result in $i - 1$ service rollbacks for P. The average number of service rollbacks T_{avg} is thus:

$$T_{avg} = \frac{0 + 0 + 1 + 2 + ... + (m - 1)}{m + 1} = 1/2(\frac{m(m - 1)}{m + 1})$$

□

Table 6: Comparative theoretical analysis of methods A, B and C

	A	B	C
Minimum	$(m/2)$	$(1/2)(\frac{m(m-1)}{m+1})$	$(1/4)(\frac{m(m-1)}{m+1})$
Average	$(n/2)(m/2)$	$(n/4)(\frac{m(m-1)}{m+1})$	$(n/8)(\frac{m(m-1)}{m+1})$
Maximum	$n(m/2)$	$(n/2)(\frac{m(m-1)}{m+1})$	$(n/4)(\frac{m(m-1)}{m+1})$

Our approach C creates a constraint-aware plan and verifies the constraints of the plan in a more efficient way to minimize service rollbacks by moving the constraints earlier in the plan.

Lemma 3. *Let $P = \{w_1, w_2, .., w_m\}$ be a constraint-aware plan of m service-nodes. Let $C = \{c_1, c_2, .., c_m\}$ be a set of constraints defined over P. In average, the number of service rollbacks using our approach is $1/4(\frac{m(m-1)}{m+1})$.*

Proof. For a service-node in position i, its constraints could be verified in positions from 0 to $i-1$ in the constraint-aware plan. Therefore, on average it could result in $\frac{0+1+...+i-1}{i}$ rollbacks. The average number of service rollbacks T_{avg} considering all the service-nodes in the plan is thus:

$$T_{avg} = \frac{0 + 0 + \frac{0+1}{2} + \frac{0+1+2}{3} + ... + \frac{\frac{m(m-1))}{2}}{m}}{m+1}$$

$$T_{avg} = \frac{1/2(1 + 2 + 3 + ... + \frac{m(m-1)}{m})}{m+1}$$

$$T_{avg} = 1/4(\frac{m(m-1)}{m+1})$$

□

Table 6 shows the minimum and the maximum number of service rollbacks and service-node calls when the execution algorithm uses n different plans to execute a task (with n plans of m component services). The maximum is the average service rollbacks in the worst case scenario, when all plans fail and the requested task cannot be executed by any plan.

Based on what we analyzed in Lemma 1, Lemma 2 and Lemma 3, the average number of service rollbacks in our proposed approach is 50% of the number of service

Table 7: Complexity of constraint adjustment algorithm

	Best	Average	Worst
computation complexity	nα	$\frac{n(n+1)\alpha}{4}$	$\frac{n(n+1)\alpha}{2}$

rollbacks in approach B. If the maximum number of constraints in each constraint set is α and the maximum number of services in a plan is n, which is the number of services in the repository, Table 7 depicts the complexity of Algorithm 5 in minimum, average and maximum cases. In minimum case, there is no constraint that can be moved back through the constraint-aware composite plan, which makes the complexity $O(n\alpha)$. However, in the worst case, all constraints of all services in the plan could be moved back to the beginning of the plan ($O(n^2\alpha)$).

In addition we evaluate the effectiveness of our approach based on real dataset using TestsetGenerator2009 [14]. Each dataset contains a WSDL file which is the repository of all generated web services. An OWL file lists the relationship between "concepts" and "things". WSLA file describes QoS values of services. To test the effectiveness of our approach, we generated 14 different test sets using the generator. The number of services varies from 3500 to 4500, and the number of concepts varies up to 10000 accordingly. Each web service has different input and output parameters. The number of input and output parameters varies from 2 to 10 and the length of generated composite services varies from 5 to 14. In addition, since the generated data using this generator is not oriented to service composition considering constraints (C) and effects (E), in the following experiments we need to modify data sets by adding a set of effects to different services to meet our experimental needs. As we discussed in Section 3.3, E represents the set of parameters whose values are affected as a result of the execution of the service. Therefore, for each service we consider the set of output parameters as the set of E. In addition, for each service, we generate the set C which is simply a set of boolean variables that assign a boolean value to a parameter in E. Then to compare the effectiveness of our approach, we compare the worst case scenario when we execute the plan n different times and n is also the total number of all constraints in the plan. In each execution, one of the constraints

fails the verification and fails the execution of the composition plan. Finally, we run our algorithm and calculate the total number of failures our approach can save compared to [92] which is approach B. Figure 9 depicts the number of rollbacks in plans that are going to be prevented as a result of our approach compared to current regular verification approach. It shows, our approach saves between 38% to 57% of executions that are failed by [92]. In addition, Figure 10 shows the time overhead that the constraint verification adjustment method adds to the service composition time. It shows that for a longer plan, it takes more time to create the constraint-aware plans.

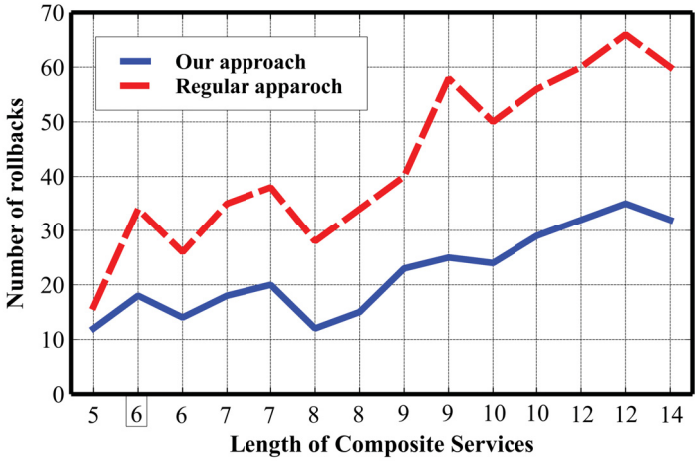


Figure 9: Number of prevented rollbacks - our approach (C) vs. regular approach (B)

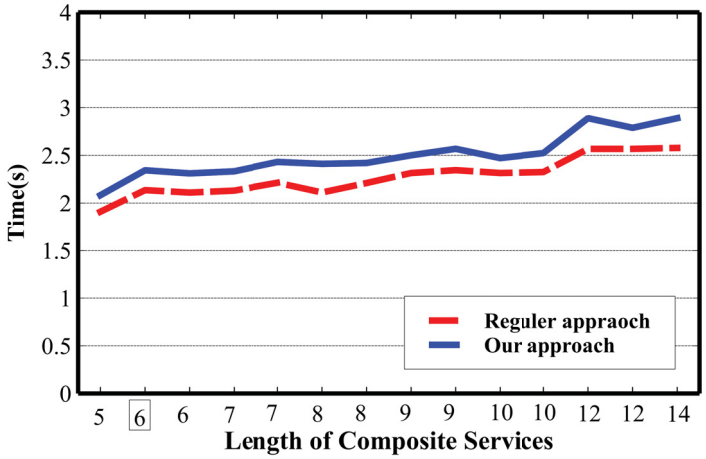


Figure 10: Constraint adjustment overhead - approach (C) vs. regular approach (B)

3.6 Summary

Constraints express limitations on service features that need to be considered during design and execution of composite services. They can be defined as required limitations of a requested service by costumers (i.e., customer constraints). Besides, most available services in real-world have constraints imposed on them by their providers (i.e., service constraints). Collectively, these constraints specify the conditions that must be met to ensure the correct execution of all involved services that collectively meet the user requirements. The verification of component service constraints has a significant impact on correct design and execution of composite services. In this chapter, we defined a model which includes definitions for user and service constraints, and their underlying concepts. Then, we proposed an approach which is aware of its component services' constraints during the execution and catch upcoming failure and thus avoid some of the incurred rollbacks. In fact, the proposed approach keeps track of the context and the constraints, and verify all related constraints as soon as the context changes. In this way, the upcoming failure can be caught sooner and the number of service rollbacks could be minimized.

Chapter 4

Runtime Constraint-aware Failure Recovery for Web Service Composition

In Chapter 3, we discussed how the set of internal constraints of a composite service is derived from the union of all constraints of composed services. These constraints should be verified to ensure services' correct execution. Indeed, the verification of some individual services' constraints depends on the values that are going to be provided by users or other services during the execution of a composite service. In this situation, if the restrictions that are set by these constraints will not be met at execution time, the service execution fails and consequently fails the execution of the composite service. In Chapter 3, we proposed an efficient constraint verification approach to verify service constraints *during* execution of a composite service. This approach can reduce the number of rollback penalties which are imposed to the system as a result of a failure in verification of service constraints in a single execution of a composite service to accomplish a task. Therefore, a recovery approach is required to complete execution of the task.

Many failure recovery approaches have been proposed to manage and recover failure in the execution of composite services [32]. Web service transaction (WST) approaches [27, 96, 29] use recovery mechanisms, including forward and backward

recovery. Forward recovery attempts to reach the original goal of the composite service by retrying or replacing components and continuing the process. Backward recovery is essentially a form of rollback that unrolls the transaction and tries to find another solution. However, these approaches are not efficient as they still impose many rollbacks during the recovery process. In this chapter, using our constraint verification approach discussed in Chapter 3, we propose a novel solution to assemble a *constraint-aware composite package* including alternative solutions for a service composition problem.

4.1 Motivation Scenario and Problem Analysis

In Section 2.1, three constraint-aware composition plans are assembled to accomplish the shopping task. In each plan, the shipment service has its constraints based on the *ProductAddress* and *DeliveryAddress* of orders. Consider a situation where *composition* 1 is picked for shopping a book. During the execution of the composition plan, after searching the book (executing W_1), it turned out that the *ProductAddress* is in Toronto. In this case, after ordering the product and making the payment (executing W_2), the execution of the shipment service (W_3) fails and consequently fails the execution of shopping composite service. In this situation, the task of buying a book is not complete and we need to find a way to complete the task.

In [96, 32] different dynamic failure recovery approaches are discussed including backward and forward recovery approaches. Backward recovery approaches [29] need to roll back effects of executed services (W_1 and W_2) and find an alternative composite service to execute the task. However, as they do not consider service constraints, the alternative plans might also fail. For example, the best alternative plan (for composition 1) based on cost is *composition* 2, whose execution will fail as a result of the constraints imposed by W_4. Forward recovery based approaches [27] look for an alternative service with the same functionality (input/output) to repair the plan (e.g. W_4 for W_3). However, as forward recovery approaches do not consider the constraints of alternative services, the recovered plans might fail again. Current

recovery approaches do not consider service constraints and that could result in a recovered plan that fails again. They might find an alternative plan that executes the same services and fail the execution over and over or they might not even be able to recover the plan. In addition, these approaches can result in high plan distance, which is basically defined as the number of newly added services appearing in the adapted composite service compared to the original plan [31].

These recovery approaches are designed based on the idea that services in a composite service might not be available after assembling the composite service. However, it should be noticed that there is a difference between failure recovery resulting from the unavailability of component services and failure resulting from service constraint verification. When a component service is not available, it is excluded from a composite service plan by the recovery process and any plan using the service will no longer be valid. However, when service constraint verification results in failure of the execution of the composite service, the recovery process should find an alternative plan to complete the execution. This should also take into consideration that the service whose constraints have been failed in a certain context may still be valid for some following executions in a different context where its constraints may be met, hence it is not required to be excluded from alternate plans. Therefore, going through recovery process for every constraint verification failure inside a composite service could add considerable overhead to system performance during the execution of the composite service [44].

Table 8 compares the current recovery approaches from the discussed perspectives. As it is discussed, forward approaches might not be able to find the possible recovery plan as it only moves forward. It also shows both forward and backward approaches can result in high plan distance as well as high number of necessary rollbacks. In addition, both approaches are specifically designed to face unavailability of component services inside the plan.

To the best of our knowledge, there is no constraint-aware failure recovery approach that can resolve the above-mentioned issues in which we are interested.

Table 8: Comparison of failure recovery approaches

Recovery Approaches	Support constraint failure	High plan distance	Successful recovery	Rollbacks
Forward	**N**	**N/Y**	**N**	**N/Y**
Backward	**N**	**Y**	**Y**	**Y**

4.2 Chapter Methodology

In this chapter, we are motivated to use our proposed constraint verification approach (see Chapter 3) and start recovery as soon as a potential failure in a plan is caught. In this way, many unnecessary executions are prevented and the number of rollbacks is reduced. In addition, considering differences between failures resulting from constraint verification failure and failures resulting from the unavailability of services inside a plan, current solution for composite services result to wastage of computational resources. We believe that constructing a new structure which includes more than one constraint-aware composite plan for a composite service request can be a solution. In this way, as soon as a potential failure is caught, the new structure should be able to switch to alternative constraint-aware composite plans during the execution and thus increase the chance of successful execution. In addition, if constraint-aware composite plans have common services, the execution of those services might be saved an prevent some rollbacks. The structure should be a constraint-aware composite service itself. However it needs specific algorithms to be designed and executed. Following the model and algorithms we proposed in Chapter 3, we design graph plan-based algorithms to define and generate the new structure. New algorithms to execute services based on their order in the new structure needs to be designed. For experimental evaluation, we implement our proposed algorithms to construct our proposed structure for each composite service request. Then, we use a publicly available data set generator to show the effectiveness of our approach compared to two common forward and backward recovery approaches.

4.3 Failure Recovery in Web Service Composition

Considering all problems and differences related to failure recovery resulting from constraint verification failure in web service composition, current failure recovery approaches can result in unnecessary rollbacks and high plan distance during the recovery process. In this section, we discuss our novel runtime constraint failure recovery approach for web service composition.

4.3.1 Composite Package Creation

All forward and backward recovery approaches [27, 96] discussed in Chapter 2 add or remove new services to the broken solution plan, while adding a new service could result in a need to repeat the constraint adjustment process. As a result, in our solution we propose the notion of **constraint-aware composite service package** to manage failure recovery in a way to reduce the number of rollbacks.

Definition 9. *A **Constraint-Aware Composite Service Package** (CaCSP) is a constraint-aware plan including constraint-aware plans that can accomplish the same task.*

Figure 12 depicts a CaCSP that includes the constraint-aware plans discussed in Section 3.1. To create a package, an algorithm is developed to integrate a subset of all alternative constraint-aware plans into a package. We use the following operators discussed in [35, 95] to describe the workflow of a constraint-aware plan.

- $\rightarrow$: Is an operator representing that the second service-node is executed when the execution of the first service-node is finished.
- $\oplus$: Is an operator representing that the two service-nodes are executed simultaneously.
- $\otimes$: Is an operator representing that one of the two service-nodes is selected to be executed.

For example, in Figure 4.11(a) *composition 1* described using the above operators. To make it more clear, in Figure 11 we did not show the constraints sets of every service-node and we only showed service-nodes. To create a CaCSP, at first we add a service-node ($W_0 =< C_0, s_0 >$) to all the plans to make all the plans to have the same starting point. $s_0 =< I_0, O_0, C_0, E_0, QoS >$ is a service where $I_0 = \emptyset$ and $O_0 = R.I$ where $R.I$ is the set of input parameters of the service composition request. It should be noted that W_0 is not an actual service-node. It only clarifies the starting point of the package graph to make creation and execution point of the package clear. To combine possible constraint-aware plans in a CaCSP, we start with a plan with the highest utility score value and then gradually add other plans to build the CaCSP. In Algorithm 6, to make the CaCSP, constraint-aware plans (like $p = W_1 \rightarrow W_2 \rightarrow ... \rightarrow W_n$) need to be converted to a format in which they only have $\rightarrow$ operator and each W_i could be a combination of service-nodes which can be executed in parallel ($\oplus$) or individual ($\otimes$). For example, *composition* 3 can be depicted as $W_1 \rightarrow W_V \rightarrow W_7$ when $W_V = W_5 \oplus W_6$.

If p_i and p_j are two constraint-aware plans such that:

$$p_i = W_0 \rightarrow W_1 ... \rightarrow W_k \rightarrow W_{k+1} \rightarrow .... \rightarrow W_x$$

$$p_j = W_0 \rightarrow W_1 ... \rightarrow W_k \rightarrow W_{k+1} \rightarrow .... \rightarrow W_y$$

and we have:

$$V_i = W_{k+1} \rightarrow .... \rightarrow W_x$$

$$V_j = W_{k+1} \rightarrow .... \rightarrow W_y$$

then, these two plans are combined in a new plan:

$$p = W_0 \rightarrow W_1 ... \rightarrow W_k \rightarrow (V_i \otimes V_j)$$

Algorithm 6 gets a set of constraint-aware plans and creates a CaCSP out of these plans. It starts with considering the first constraint-aware plan as a CsCSP. Then, in

each step, it adds a new constraint-aware plan to that. Every time a plan needs to be added, the intersection of the plan with the package should be found (line 3-12). Then, based on what we discussed, they should be combined (line 13-16). Figure 4.11(c)

Algorithm 6 Composite Package Creation

Input: *planSet* (Set of constraint-aware plans)
Output: *pkg_plan* (a constraint-aware composite service package)

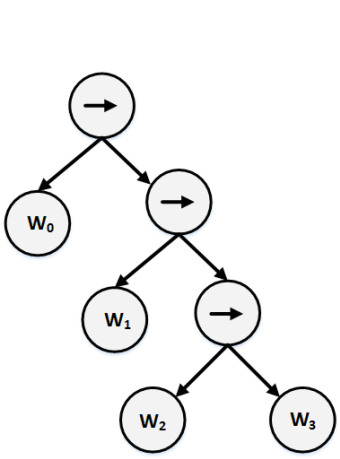
```
1: pkg_plan = a constraint-aware plan from planSet
2: planSet = planSet − pkg_plan
3: for (each plan ∈ planSet) do
4:    i = 0
5:    repeat
6:       for (each serviceNode ∈ pkg_plan) do
7:          if (serviceNode ∉ plan) then
8:             break
9:          end if
10:      end for
11:      i = i + 1
12:   until (i ≤ pkg_plan.length)
13:   l1 = partialPlan(i + 1, pkg_plan.length)
14:   l2 = partialPlan(i + 1, plan.length))
15:   tempPlan = l1 ⊗ l2
16:   pkg_plan = pkg_plan.part(i) → tempPlan
17: end for
18: return pkg_plan
```

shows how a CaCSP is being created based on alternative constraint-aware plans for the shopping composite service.

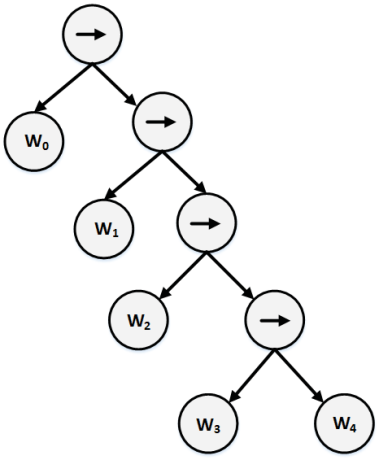
4.3.2 Composite Package Execution

As it is depicted in Figure 12, the structure of a constraint-aware composite service package is different from the structure of a simple composite service. Therefore, it cannot be executed like a regular constraint-aware plan that executes services in a specific order. In the following paragraph, we define the required concepts and then discuss the CaCSP execution algorithm in detail.

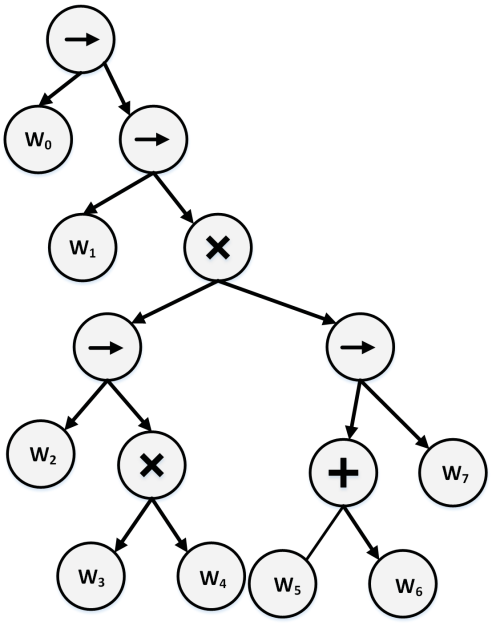
In AI planning for AWSC, a web service alters the state of the composite service upon execution. When a composite service is being executed, the state of the



(a) CaCSP of composition 1.



(b) CaCSP of composition 1 & 2.



(c) CaCSP of three alternative plans.

Figure 11: Step-by-step results of CaCSP creation process.

composite service changes step by step by execution of each component service. The composite service execution ends when all component services have been executed and the output of the final service in the plan is returned as the result of the execution of the composite service.

Definition 10. *A **State** is the set of all $<$ ontologyType $\mapsto$ value $>$ mappings representing the values associated with features of the component services in a constraint-aware plan, each of them being initially assigned NULL values. The state of a service represents the **Context** of the service.*

The state of a constraint-aware composite plan is a set of mapping where the list of features in the data model of composite service is assigned with values during the execution of the composite service. At each step of the execution, the state represents the current context of the execution. We also need to define the way that a service-node changes a state value by its execution and in which condition a service-node can be applicable to a state.

Definition 11. *A service-node ($W =< C_S, service >$) is **applicable** to a state $S = \{< T_1 \mapsto v_1 > ... < T_n \mapsto v_n >\}$ (where $\{T_1, T_2, ..., T_n\}$ is a set of ontology types representing all features in the data model of the constraint-aware plan, and $\{v_1, ...v_n\}$ are literal values of the same respective types) denoted as $S \succ W$, if verification of all constraints in C_S would be satisfied based on the values assigned to the features in S.*

In classical planning, a **State Transition Function** (γ) is a function that applies effects of execution of a service on a state, if the service is applicable. In our model, when a service-node ($W =< C_S, service >$) is applicable to state S, a **State Transition Function** (γ) is applied to change the state of execution to S' : $S' = \gamma(S, W)$.

Definition 12. *A **State Transition Function** (γ) is a function that applies effects of execution of a service-node(W) on a state (S), if the service is applicable ($S \succ W$).*

It should be noted that if all service-nodes of a constraint-aware composite service are composed in sequential order like *composition 1* in the motivation scenario, the

Table 9: Constraints verification plan

	C_{s1}	C_{s2}	C_{s3}	C_{s4}	C_{s5}	C_{s6}	C_{s7}
Constraints	c_1, c_2, c_{31} c_{41}, c_{71} c_5, c_6	c_{32}, c_{42}, c_{72}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

goal state is calculated as $G = (\gamma(\gamma(\gamma(S_0, W_1), W_2), W_4)$. However, if services are composed in parallel order (like W_5 and W_6 in *composition* 3), the goal state of the composition is calculated as: $G = \gamma(\gamma(\gamma(S_0, W_1), W_5) \bigcup \gamma(\gamma(S_0, W_1), W_6), W_7)$. Since a CaCSP has a different structure from a solution plan, we provide a different execution mechanism for it. Figure 12 shows the CaCSP of the scenario discussed in Section 3.1. The general idea behind the package execution is to execute plans inside the package one by one. During the execution of each plan, if the verification of a service-node constraint fails, the execution system prunes all plans that are related to the failed constraint. Then the execution continues with one of the remaining plans. For example for the package of our discussed scenario (Figure 12), the execution starts with the first service-node (w_1) of *composition 1.* Before the execution of w_1 all constraints moved before w_1, including $c_1, c_{31}, c_{41}, c_{71}$ (Table 9), are verified. Then, if the verification of any of them fails, the package will prune all plans related to the failed constraint. For example, consider the case where c_{31} fails the verification before execution of w_1. It means that, based on the delivery address of the shopped item, the item cannot be shipped using w_3. As a result, any plan that includes w_3 (e.g. *composition 1*) is pruned from the CaCSP. This process will continue until all plans are pruned from the package or at least one plan completes the execution.

Algorithm 7 represents the recursive approach that is designed to execute a CaCSP. The execution starts from the root and proceeds based on the service-node or the operator in the root. To execute a service-node (or set of service-nodes) in the root, the algorithm first verifies the set of constraints of the service. If the verification of all constraints succeeds, it executes the service-node. However, if the verification fails, the CaCSP should be pruned.

In addition, if there is an operator in the root, the algorithm should make the right

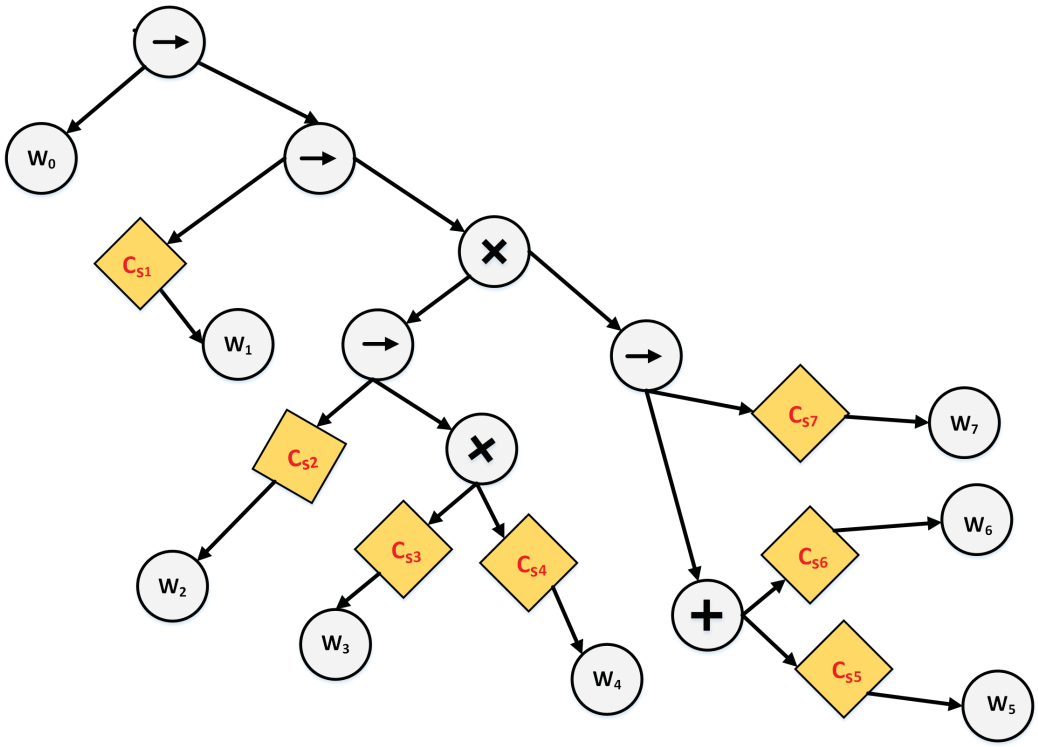


Figure 12: Constraint-aware Composition Service Package (CaCSP)

decision to continue the execution of the plan (lines 15-32). To do that, it starts with computing the left and right sub-trees of the composition tree (lines 1-14). Then, based on the operator in the root, it continues the execution. For $\rightarrow$ first, the left sub-tree should be executed. Then, after it is finished successfully, the right sub-tree is executed. Operator $\otimes$ represents the situation that execution of one of the left or right sub-trees of the package is enough to complete the execution (lines 22-27). Finally, for operator $\oplus$, both sub-trees must be executed in parallel (lines 28-31). It should be noted that, in our system, we made an assumption that service-nodes that are going to be executed in parallel cannot affect the same parameters in the CaCSP data model.

In case the verification of a service constraint fails during the execution of a package, the execution algorithm prunes the package. Then, it checks whether the package is still a valid package or not (Lines 10-14). A package is valid for execution if, after pruning a failed plan from the package, it still has alternative plans to complete the execution. Therefore, if the package is valid the execution algorithm

Algorithm 7 cmp_pkg_execution

Input: pkg_plan (a constraint-aware composition package), S_0 (initial state of execution)

Output: either execution state state or NULL

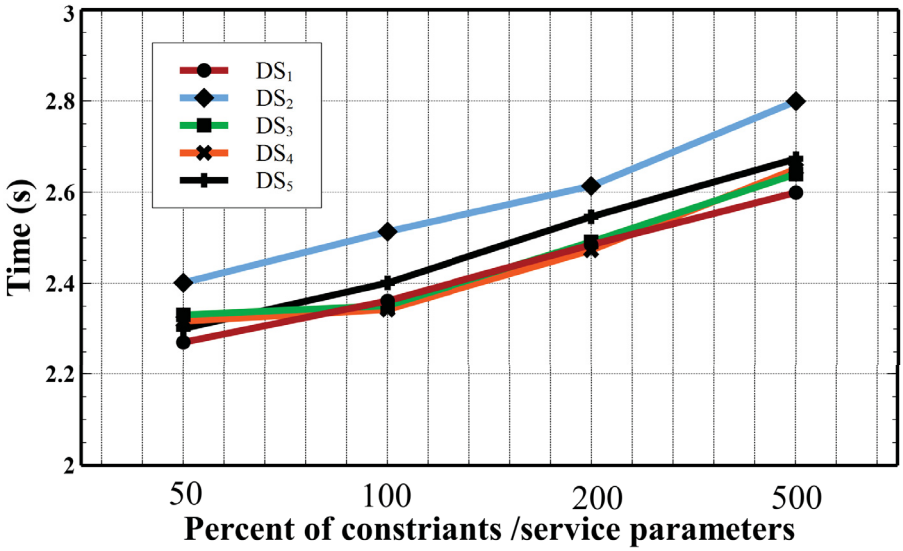
```
1: result = Null
2: if (IsServiceNode(pkg_plan) then
3:    if (GSD(pkg_plan.C_S), S_0) then
4:       StateList[pkg_plan] = γ(S_0, pkg_plan)
5:       return StateList[pkg_plan]
6:    else
7:       Prune(pkg_plan)
8:       if IsValid(pkg_plan) then
9:          return cmp_pkg_execution(pkg_plan, S_0)
10:      else
11:         return Failure
12:      end if
13:   end if
14: else
15:   operator = The operator in the root of the composite package
16:   t_1 = leftSubTree(pkg_plan, operator)
17:   t_2 = rightSubTree(pkg_plan, operator)
18:   if (operator is →) then
19:      result = cmp_pkg_execution(t_1, S_0)
20:      result = cmp_pkg_execution(t_2, result)
21:   end if
22:   if (operator is ⊗) then
23:      result = cmp_pkg_execution(t_1, S_0)
24:      if (result is Null) then
25:         result = cmp_pkg_execution(t_2, S_0)
26:      end if
27:   end if
28:   if (operator is ⊕) then
29:      result = cmp_pkg_execution(t_1, S_0)
30:      result = cmp_pkg_execution(t_2, result)
31:   end if
32: end if
33: return result
```

continues until it returns the final execution state (including the execution results) of the package, unless it returns failure which means the package cannot complete execution of the task using any of alternative composition plans.

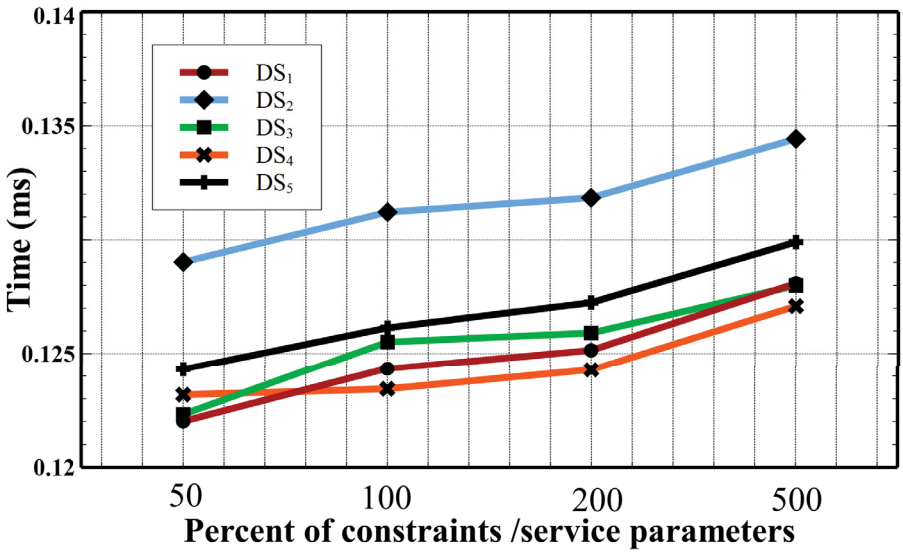
4.4 Analysis and Experimental Results

Our approach has two overheads compared to other approaches: package creation at composition time and constraint verification at run time. In Chapter 3, we discussed the overhead that constraint verification adjustment could add to service composition time for a composite service request. However, we believe the number of constraints of component services is also important with the more constraints inside the plan, it takes more time to adjust constraints at composition time and verify constraints at execution time of a package. Figure 13 depicts package creation and constraints verification time inside a composite package. For this experiment, we randomly generated 5 different datasets (DS_1, DS_2, DS_3, DS_4, DS_5) using the WSC 2009 Testset Generator [14]. Each dataset contains a WSDL file which is the repository of web services. An OWL file lists the relationship between "concepts" and "things". The number of services for each dataset is around 4000, and the number of concepts varies from 3000 to 3500 accordingly. Figure 13 discusses package creation and constraints time inside a composite package when the number of constraints for each service is 50%, 100%, 200% and 500% times more than the number of service parameters. Figure 4.13(a) compares the package creation time for all datasets. In addition, Figure 4.13(b) discusses the verification time overhead that is added to the execution time of each package. The generated package using DS_2 has the most number of services among all generated packages, while other packages have almost the same number of services. Therefore, it is clear as the number of constraints increases, the time for package creation and constraint verification is increased as well.

We also compare the results of our proposed constraint-aware composite service package approach with other failure recovery approaches including re-planning and



(a) Packaging creation time



(b) Constraints verification time

Figure 13: Package processing overhead

repair (Chapter 2) which are two well known forward and backward recovery approaches. For this experiment, we use the above five randomly generated datasets (DS_1, DS_2, DS_3, DS_4, DS_5). Then, we randomly failed services (service-nodes) inside the solution plans in our generated CaCSP. Different approaches were compared to see how many rollbacks were imposed as a result of the failure recovery. In repair, if the plan cannot be recovered, all the services until the broken point need to be rolled back. Each point is obtained from the average of 3 independent runs which in total is 15 different runs.

We compared all approaches from three aspects including the number of rollbacks (Figure 14), the computation time (Figure 15) and the plan distance (Figure 16). Figure 14 depicts the results of our experiments in terms of the number of rollbacks. It shows that re-planning imposes more rollbacks than other approaches. The reason is that every time a failure happens, re-planning needs to design the solution from the beginning. It is also clear that our approach imposes the fewest number of rollbacks compared to other approaches. This is due to the fact that our approach potentially reuses partially executed parts that are common between the current failed plan and its alternative selected after the failure. It also allows to predict some failures that are going to happen later and to avoid going forward on a constraint-aware plan that we know is going to fail, thus saving rollbacks by *predicting* failure.

We also compared all approaches based on the computation time required to proceed with failure recovery (Figure 15), i.e., the time that the algorithm requires to do the recovery. Re-planning has the worst time as it is the same as running the composition algorithm from the beginning after excluding failed services from the repository. In addition, the performance of our approach is not significantly different from repair. Finally, Figure 16 makes a comparison based on the plan distance. It shows re-planning has the highest plan distance which is obvious as re-planning basically designs a new composite service with new services. Our approach, in general, has less plan distance than repair. This is because in case the repair is not successful, it results in a plan distance as worse as re-planning.

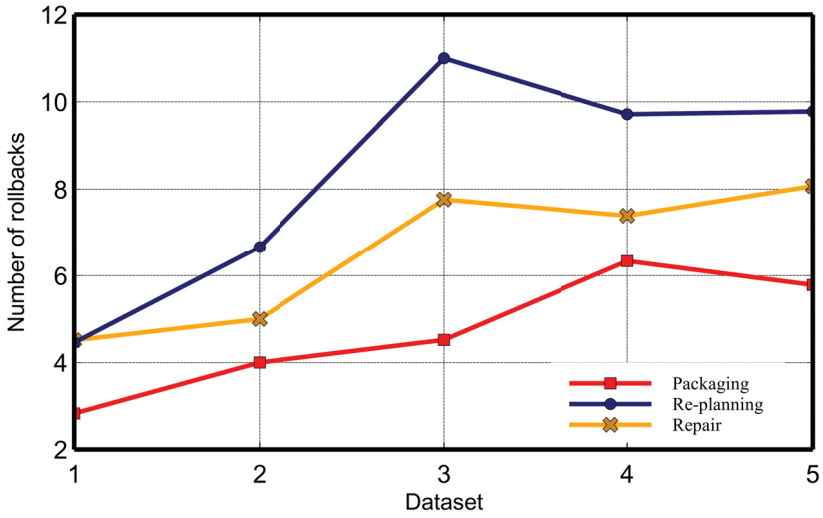


Figure 14: Number of rollbacks

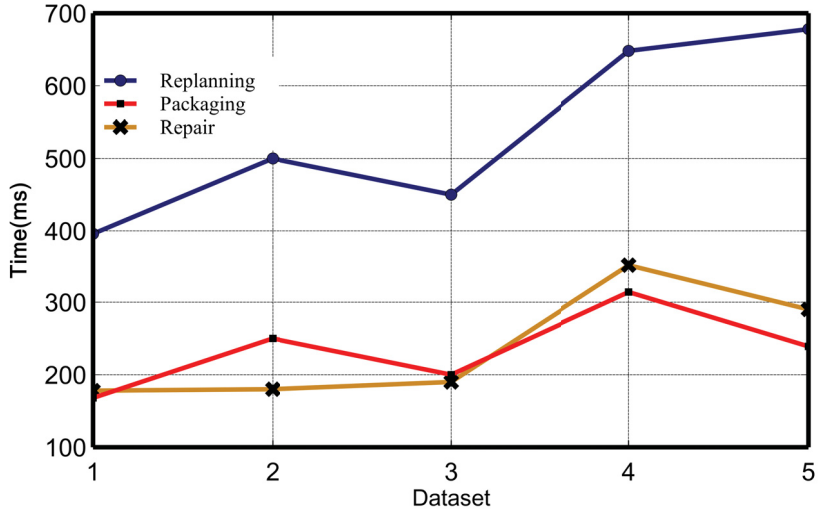


Figure 15: Time performance of different approaches

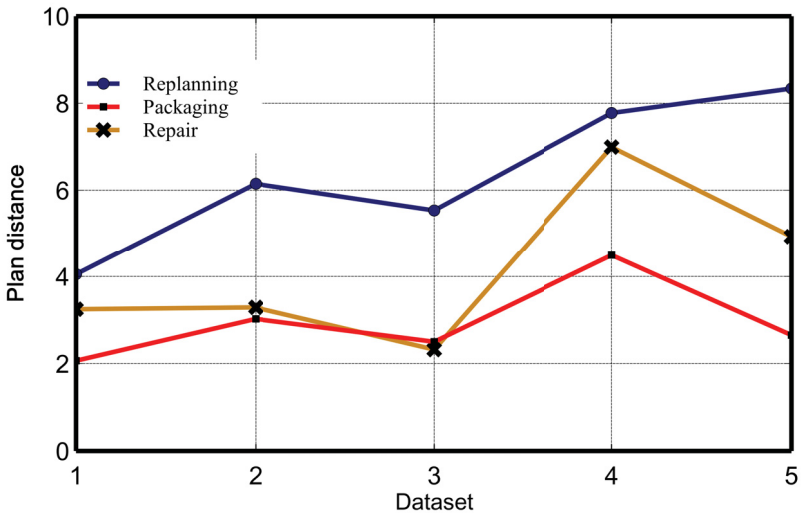


Figure 16: Plan distance of different approaches

4.5 Summary

There is a difference between service execution failure resulting from unavailability of component services and failure resulting from service constraint verification inside composite services. Current service composition failure recovery approaches result in high plan distance and wastage of computational resources. In this chapter, we proposed a constraint-aware failure recovery approach which uses our proposed verification approach in Chapter 3 to catch upcoming failures and start recovery as soon as possible. In our approach, we proposed a new structure called Constraint-aware Composite Service Package (CaCSP). It includes more than one constraint-aware composite plan for a composite service request. In this way, as soon as a potential failure is caught for one executing plan in CaCSP, it switches to other plans. We compared our approach with two other failure recovery approaches from different perspectives. The experimental results demonstrate that our approach provides a better solution regarding reduction of plan distance as well as the number of imposed rollback penalties during the recovery process.

Chapter 5

External Constraints

As we discussed in earlier chapters, most services in the real world are not universally applicable. Some services impose certain conditions or restrictions (i.e. constraints) which are defined by their providers [92]. Such constraints specify the conditions that must be met to ensure correct execution or proper interaction with the service [92]. We called them the *internal constraints* of a composite service as they belong to providers of services inside the plan. Internal constraints are not the only constraints that may need to be considered at execution time of a composite service. Even after the initial assembly and deployment of the composite service, emerging constraints might be imposed on the composite service and its component services. For example, consider a shipment service which can only ship items from/to North America. This service is used in a composition of some services to build a shopping composite service. After the shopping service has been assembled and deployed, new international regulations might come to make the company change its business rules and stop the clients to ship any item to/from the United States. Additionally, such an external constraint might also be lifted or re-applied later in the future. Considering the shipment service constraints (ship only to/from North America) and the newly added constraint (not to ship to/from the United States) makes the composite service to only let the users shop to/from any address in North America except the United States. It is important to note that this newly defined constraint is not a limitation given by a service provider nor a service requester. It is a constraint that comes externally

and it needs to be applicable after the composite service has been assembled and deployed, i.e. it requires dynamic injection of the constraint in the composite service. In addition, such a constraint does not change the composite service input/output specifications and only puts more restrictions on the composite service usage. We call these limitations *external constraints*, compared to service requester constraints and internal constraints.

Most available service composition approaches do not consider such external constraints during the composition process, nor during the execution of composite services. The few existing approaches that deal with constraints only focus on service (i.e. internal) constraints [38, 93, 92, 46]. In a highly dynamic environment, external constraints might be defined and applied dynamically. Composite services should be able to adapt to them immediately when they apply, and similarly adapt to the removal of the constraints when they cease to apply. Adaptation in web service composition has been under attention in response to a service failure or new requirements that can result in the change of the composite plan structure and specifications. In the situation described above, constraint adaptation does not add new features (input/output, QoS) to the composite service. It only affects composite services by adding more usage restrictions. In a highly dynamic environment, many constraints might be created or disappear, or apply or cease to apply at any time. Therefore, using current adaptation approaches such as repair and re-planning has some drawbacks [99]. First, re-planning usually requires time costs similar to the original planning process needed. Thus, re-planning is not very efficient in general. Second, both approaches abandon some parts of the existing plan (even the whole plan in some cases). Therefore, the resulting plan can be a very different plan with new services and specifications from its predecessor. This might not be acceptable in the real world since users often sign contracts with web services providers, which is defined according to the original plan. Changing the composition plan means to redefine, renegotiate, or often abandon existing business contracts. Additionally, changing a composite service plan will most often result in changing the data model used by a composite service, which might be highly problematic if a composite service

is expected to retain all transactional data of service usage.

In this chapter, we focus on the problem of constraint adaptation in web service composition when external constraints impose additional restrictions after the deployment of composite services.

5.1 Motivation Scenario and Problem Analysis

One of the difficulties in the shopping scenario we discussed in Chapter 3 was that the shopping process needs to take into account any internal constraints that can affect any step of the shopping process as defined in general and eventually used in different operational usage contexts. Even more difficult and interesting is the possibility of having an existing shopping process' usage be imposed with and adapted to some externally defined constraints that may come into existence after the service has been assembled and deployed. Consider a shopping service based on services in Table 10 that consists of the following sequence of tasks: searching for products, submitting an order, paying for the order, and shipping of the order. The service composition algorithm designs a composite plan (Figure 17) for shopping based on the services expressed in Table 10.

As we discussed in earlier chapters, service constraints can affect the execution of the respective services and, by extension, they also affect any composite service using these services (Figure 17). For example, based on the constraints defined

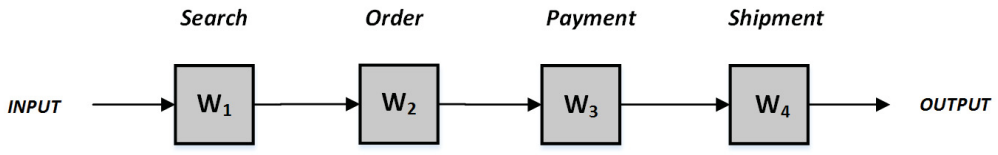


Figure 17: Shopping composite service

at the level of the shipment service, the composite shopping service can only ship items to/from North-America. In addition to service constraints defined by the services used in the composite shopping service, there might also be additional restrictions that are required to be considered after the composite service has been

Table 10: Available services

#	Service	Input	Output	Constraints
W_1	Search	ProductName, DeliveryAddress	ProductNumber, ProductAddress, PaymentAmount	$C_1 = \emptyset$
W_2	Order	ProductNumber	OrderNumber,	$C_2 = \emptyset$
W_3	Payment	PaymentAmount, PaymentMethod	PaymentConfirm	C_{31} = PaymentMethod $\in$ {Visa, MasterCard} C_{32} = PaymentAmount $\leq$ 10000
W_4	Shipment	PaymentConfirm, DeliveryAddress, ProductAddress, OrderNumber	ShipmentConfirm	C_{41} = ProductAddress $\in$ North-America C_{42} = DeliveryAddress $\in$ North-America

assembled and deployed. Here are three examples: (1) Due to a failed deal with Master Card, the shopping store may want to stop accepting Master Card as an accepted payment method (p_1) for the shopping store, even though the payment service it uses is potentially accepting it. (2) Due to some problems in the delivery system at the Canada/United-States border, the shopping application may want to momentarily stop accepting any shipment from/to the United-States until the problem is resolved (p_2). (3) Based on newly adopted Quebec government rules, any purchases made to an address in Quebec needs to pay an additional sales tax (p_3). Note that in none of these cases are the individual services aware of the constraint, nor are they responsible for it. It is clear that these additional requirements include some constraints that have to be followed by required actions. The adaptation of the composite shopping service to these external/additional constraints requires additional factors to be considered during the execution the composite service, even though the individual component services are not aware of them, nor are they responsible for them. First, it is important to note that none of these limitations require the modification of the composite service plan that is build based on input, output and required QoS features. Second, each of these restrictions can affect the execution of the composite service as a whole, but does not require a change in the definition of the services used inside the plan. Therefore, these externally defined constraints are to be applied, resulting in an action applied at a specific position inside the composite service. For example, p_1 and p_3 impose constraints that need to

be considered before the execution of the payment service.

Following this idea, the composite service plan for our example scenario can be constructed as depicted in Figure 18. In this situation, in a constraint-aware plan, internal service constraints are verified before execution of services inside the plan.

5.2 Chapter Methodology

As external constraints do not change the composite service input/output specifications, our strategy is to first update our composite service model (Chapter 3) to express externally defined constraints. Then, we redefine the constraint-aware composite service structure in a way to adapt to these externally defined constraints at runtime without re-construction of the composite plan. However, external constraints have to be added to a specific position inside composite plans. Therefore, to find the effective insertion points of external constraints inside a plan, we design graph plan-based algorithms to add external constraints into composite plans based on the data model of services inside a composite service. For evaluation, we compare the time performance of our adaptation approach with current web service composition adaption approaches. Current adaptation approaches re-construct the composite plan for any new constraints that needs to be added to the composite service, which requires to repair or re-build the composite service using services that are defined with the new constraints, thus adding them to the resulting plan, but with the definite disadvantage of resulting in a different composite service. We implement our proposed algorithms and use a publicly available data set to compare the performance of our approach compared to current adaptation approaches.

5.3 Composite Service Constraint Adaptation

Considering problems discussed in Section 5.1, we propose a better solution which is to define a composite service model that embeds the adaptation to these externally defined requirements at runtime without re-constructing the plan Figure 18. In this

Table 11: Table of policies

	C_p	E_p
p_1	$\{PaymentMethod \notin MasterCard\}$	$\{PaymentConfirm\}$
p_2	$\{ProductAddress \notin USA,$ $DeliveryAddress \notin USA\}$	$\{ShipmentConfirm\}$
p_3	$\{DeliveryAddress \in Quebec,$ $PaymentAmout \neq NULL\}$	$\{PaymentAmount\}$

situation, in a constraint-aware plan, internal service constraints are verified before execution of services inside the plan. In addition, external constraints are to be dynamically added to the plan to apply the required adaptation to externally defined constraints.

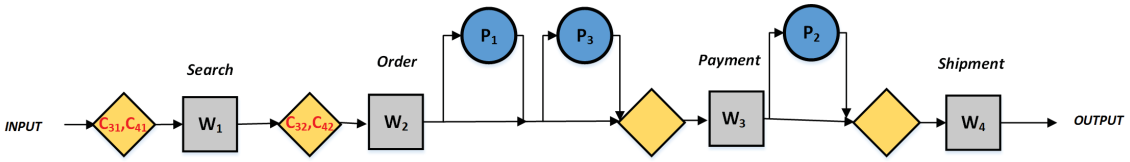


Figure 18: Constraint-aware composite service plan.

To implement that, we define the notion of *policy* to formally express external constraints as discussed in Section 3.1. Policies express a condition, which is a set of constraints and can be followed by a set of actions. In our model, we express the set of actions by a set of parameters that can be modified as a result of applying the policy.

Definition 13. *A **policy** is a tuple $p =< C_p, E_p >$ in which C_p is a set of constraints and E_p is a set of parameters that the policy modifies.*

Definition 14. *A **policy-based plan** is a constraint-aware plan in which each node is a tuple $< P, W >$ where W is a service-node and P is the set of all policies that are to be applied before W in the plan at runtime.*

Now based on our definition of the *policy*, all policies defined in Table 11 can be added in the policy-based composite service depicted in Figure 18. Figure 19 shows the policy-based composite service (based on the services presented in Table 10) when:

$$P_1 = \emptyset, P_2 = \emptyset, P_3 = \{p_1, p_3\}, P_4 = \{p_2\}$$

During the execution of a policy-based plan, before execution of each service- node all policies which are added before the service-node will be applied.

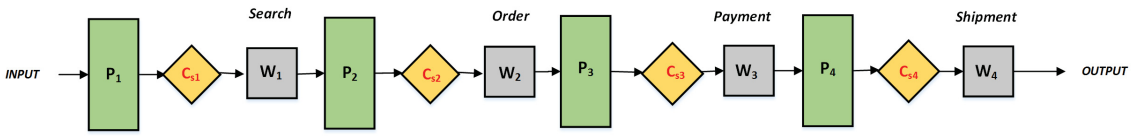


Figure 19: Shopping composite service

Algorithm 8 implements our approach to adapt a service plan according to a given policy. However, before adding a policy to a plan, it has to figure out whether or not the policy could be added to the plan. This is an important aspect since a policy should not be added to a plan if it has no effect on execution results of this plan. In addition, wherever the policy is added inside the policy-based plan, all its constraints should be verified. The set of features of a policy ($< C_p, E_p >$) includes all the features of the policy constraints ($Features(C_P)$) and the set of affected parameters of the policy ($Features(E_P)$).

$$Features(policy) = Features(C_P) \cup Features(E_p)$$

In order for a policy to be added to a plan, the set of features of the policy should be a subset of the data model of the composite service.

$$Features(policy) \subset DM(plan)$$

In addition, the proper position of the policy inside the policy-based plan needs to be found. This position is dependent on the data model of services inside the plan. A policy ($p =< C_p, E_p >$) needs to be applied before execution of a service in the plan if the data model accumulated up to its execution includes all parameters in C_p and the data model includes all parameters in E_P. To do that, the algorithm searches the composite plan and compares the data model of the services inside the plan with the set of parameters of E_p (line 3-4). Then, if the policy can be added before a service inside a plan, all required parameter values to evaluate the policy constraints

need to be available at this specific position inside the plan at runtime (line 7-15). Therefore, it considers all services that are going to be executed before the service in the plan ($preSrvSet$) and calculates all available parameters ($avlParams$) (line 6-13).

For example, suppose that p_1 is considered for addition to the composite service depicted in Figure 19. First, it should be checked whether $Features(p_1)$ is a subset of the data model of the plan. As every parameter in $Features(p_1) = DeliveryAddress, ProductAddress, ShipmentConfirm$ is $DM(plan)$ the policy could be added to the plan. Then, p_1 will be added before W_3 in the plan as $E_{p_1} \subset DM(W_3)$ and $DeliveryAddress$ and $ProductAddress$ are in $avlParams$.

Algorithm 8 Constraint Adaptation Algorithm

Input: *policy_plan* (a policy-based plan), *policy* (policy to be inserted in the plan)
Output: *policy_plan* (a policy-based plan added with the new policy, if applicable)

```
1: if (fatures(policy) ⊆ D_M(policy_plan)) then
2:   for (each < p, service > ∈ policy_plan) do
3:     preSrvSet = ∅
4:     if (p.E ⊆ D_M(service)) then
5:       preSrvSet = preSrvSet ∪ service.predecessors
6:       avlParams = ∅
7:       while (preSrvSet ≠ ∅) do
8:         serSet = ∅
9:         for (each service ∈ preSrvSet) do
10:          avlParams = avlParams ∪ service.O
11:          serSet = serSet ∪ service.predecessors
12:        end for
13:        preSrvSet = serSet
14:      end while
15:      avlParams = avlParams ∪ R.I
16:      if (p.C.features ⊆ avlParams) then
17:        P = P ∪ policy
18:      end if
19:    end if
20:  end for
21: end if
22: return policy_plan
```

5.4 Analysis and Experimental Results

In this section, we conduct two experiments to evaluate our proposed approach. The first experiment focuses on the performance of our constraint adaptation approach and the second one compares it with other web service adaptation approaches.

Our experiments have been performed on five different datasets generated using `TestsetGenerator2009` [14]. Each data set contains (a) a WSDL file, which is the repository of all generated web services; (b) an OWL file that lists the relationship between "concepts" and "things"; (c) a WSLA file that describes QoS values of the services. Table 12 represents the number of services and concepts in each data set. All experiments are performed on a PC platform with Intel CPU 3450 (2.67GHz),

Table 12: Generated datasets

	Data Sets				
	1	2	3	4	5
Concepts	1000	4015	8000	10000	15000
Services	1002	3006	5000	7001	10000

Windows 7, and 8GB RAM. The experimental platform is implemented in JAVA under the Eclipse environment. Since the datasets generated using this generator are not oriented to service composition considering constraints (C) and effects (E), in the following experiments we augmented the datasets with sets of constraints and effects for each service to meet our experimental needs. As we discussed in page 29, E represents the set of parameters whose values are modified as a result of the execution of the corresponding service. In our experiments, for each service, we consider the set of output parameters as the set E. In addition, for each parameter in a service data model, there could be a constraint that expresses limitations on a parameter. In our experiments, the number of constraints for each service is generated randomly by having between 1% to 100% of parameters defined in each service that have constraints.

5.4.1 Adaptation Algorithm Performance

The first set of experiments evaluate the performance of our approach over the generated datasets discussed in the previous section. Our service composition algorithm produces a solution plan for each generated data set. The length of the generated solution plans varies from 6 to 18 service layers. We also generated 5 different sets of policies that include 1 to 50 different policies for certain parameters applicable to the data model of the plan. We evaluate the performance of our solution by adding (injecting) policies to the composite service plan. In order to make sure policies are distributed fairly across services inside the solution plans, we generated policies in a way to make sure at least one policy will be applicable before each service execution in the plan. Figure 20 depicts the summary of results of our experiments.

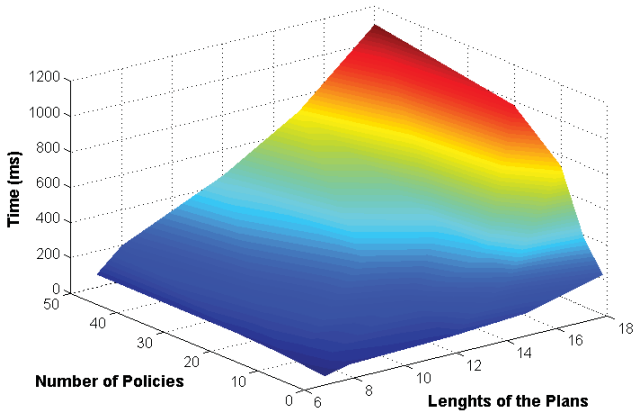


Figure 20: Constraint adaptation algorithm performance

It shows the number of policies as the x axis that varies from 1 to 50, and the length of the solution plan as the y axis, varying from 7 to 18, and the time spent as the z axis. Each point is obtained from the average of five independent runs. Then we run the algorithm for each data set using a different number of policies. The graph shows a general trend of linear growth in time when the number of policies and the length of composite plans increases.

5.4.2 Comparative Evaluation

The second set of experiments compares our approach with other web service composition adaptation approaches such as classic re-planning and repair [99]. The complexity of using repair and re-planning is exponential due to the backtracking technique used in their algorithm [99]. However, the complexity of our solution is linear according to the maximum length of the plan and the number of parameters of the services inside the plan.

For our comparative evaluation, we consider that the adaptation of a composite plan to new policies using re-planning and repair is equivalent to request a re-planning or repair after the plan has failed at any position where the policy could be added to the plan. The plan then needs to be re-constructed in a way to consider C_p and E_p for each policy ($p_1 = \langle C_p, E_p \rangle$). Web service composition repair [99] is an approach that aims at keeping as much of the composite service as possible before the breaking point, and to generate another solution from there to the goal state. For the repair algorithm, a new branch from the broken point of the plan is generated that includes C_p and E_p. As a result, the repair algorithm needs to be changed to consider the C_p and E_p sets in a generated plan. If a policy can be applied before W_j inside a plan including n services $(W_1, \ldots, W_n)$, the partial plan l, which includes all services to be executed after W_j, needs to be repaired. Then, the repair algorithm needs to generate $l' = W_k \ldots W_m$ having the following specifications: $\bigcup_{l=k}^{m} E_{l'} = \left(\bigcup_{l=j}^{n} E_l\right) \cup E_p$ and $\bigcup_{l=k}^{m} C_l = \left(\bigcup_{l'=j}^{n} C_l\right) \cup C_p$.

The re-planning algorithm is another adaptation approach that requires repeating the composition algorithm with consideration of the new constraints that the policy applies to, which is essentially the same as the planning algorithm.

We compare the constraint adaptation (i.e. policy injection) time among these three approaches (including ours). In each test we add a set of policies with 1, 5, 10 policies to the plans and evaluate the time required to compute a solution. The policies are generated based on the data model of the plans. We only define policies for which repair and re-planning algorithms can find alternative solutions to fulfill the

policies' specifications. We also make sure each service layer has at least one service parameter from the model with applicable policy. Each point is obtained from the average of five independent runs when all algorithms (repair and re-planning) can find solutions. Figure 21 compares the results of policy adaptation for 1, 5 and 10 policies using each approach. All approaches show linear growth when adding more

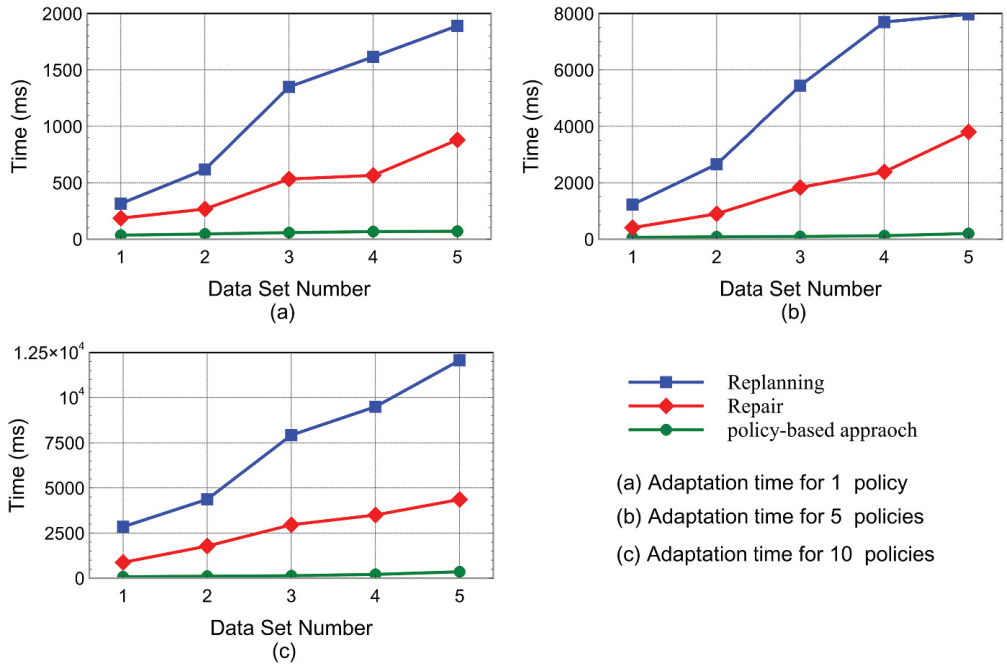


Figure 21: Policy-induced adaptation time

policies. However, re-planning shows a higher rate of growth compared to repair and to our approach. In addition, Figure 22 shows the performance of different algorithms when the number of policies increase from 1 to 50 for data set 5. As expected, the re-planning algorithm shows a higher trend of growth as it is a repetition the planning algorithm with policies added. In addition, the repair algorithm does not show a constant trend of growth (for 10 policies) as its performance is dependent on the length of the plan and the position the policies that need to be added in the plan. However, the performance of both algorithms is highly dependent on the number of services inside repositories. Compared to both repair/re-planning algorithms, our approach has a lower execution time. The reason is that our approach only acts based on the length of the composite service and it is not dependent on the size of

the service repository.

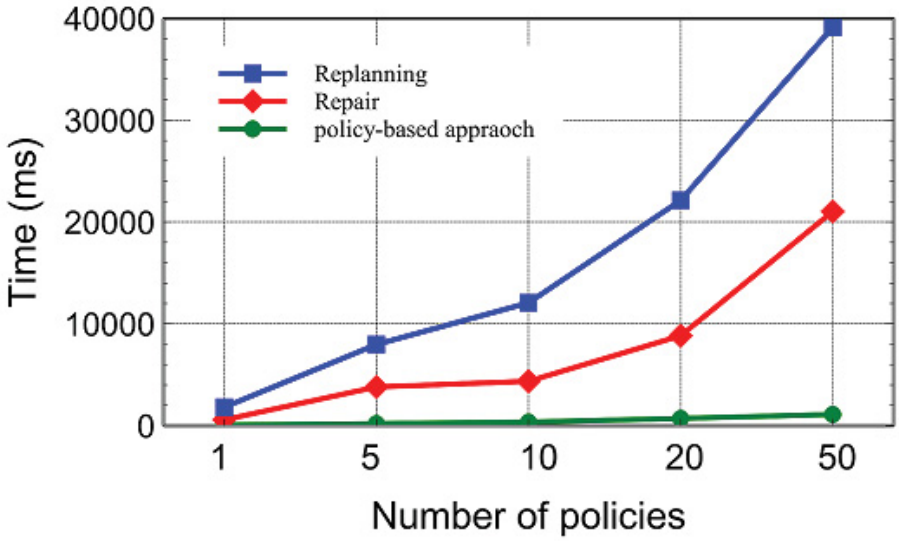


Figure 22: Adaptation performance for data set 5 based on the number of policies

From these initial experimental results and intuitive analysis of the proposed algorithms, we have learned that factors such as the location of policy injection inside the solution plans, the length of solution plans, and the size of the service repository can affect the composition and adaptations algorithms' performance.

One factor that needs to be considered in our evaluation that we did not do is the *plan distance*, which is the number of services appearing in the adapted composite service compared to the original plan [31]. For example, [101] shows that repair has a better plan distance compared to re-planning. However, as our approach does not modify services inside the plan, it has a minimum possible plan distance compared to other approaches, which is a definite advantage.

5.5 Summary

In addition to service constraints, other constraints might be imposed to put externally-defined restrictions on composite services. Such externally-defined restrictions are likely to be defined and become or cease to be applicable after the

composite service has been assembled and deployed. In this chapter, we provide a solution for adaptation of externally defined constraints in web service composition. We update our model and algorithms to generate flexible constraint-based composite services that can be adapted to constraint change after service composition. From the experimental results and intuitive analysis of the proposed algorithms, we have learned that the factors such as the location of policy injection inside the composite service, the length of resulting composite plan, and the scale of the service registry, are factors that affect the performance of the composition and adaptations algorithms. Our analysis demonstrates that the computational complexity and performance of our proposed approach is considerably better than other adaptation approaches.

Chapter 6

Policy-based Composite Package

In earlier chapters, we discussed internal and external constraints and their roles in the web service composition process and the execution of composite services. We discussed how service constraints need to be verified at composition and execution time of composite services and we proposed an efficient constraint verification approach to track the composite service context and service constraints and verify all related constraints as soon as the part of context on which they depend is changing. Then, we proposed a novel structure called a Constraint- Aware Composite Service Package to manage the recovery of constraint verification failure during execution of composite services. Finally, we proposed a solution to adapt composite services to externally defined constraints during the execution of composite services.

In the real world, different kinds of constraints can be applied to a working composite service. In addition to internal constraints, which are defined by the service providers, externally-defined constraints can also exist, i.e. constraints that are not provided by neither the service requester nor the service providers. We refer to these as *policies*. For example, policies can be defined by regulatory bodies such as governmental offices. In essence, policies are any constraints that are potentially applicable to all services, provided that they are subject to it. The application of external constraints to a web service composition inevitably interferes with the verification of internal constraints. Therefore, there needs to be a mechanism to make sure external constraints are properly injected/removed to/from a composite

service plan, and verified during execution of composite services. In this chapter, first we define the notion of a *Policy-based Composite Package* which is our solution to handle both internal and external constraints. This solution includes all approaches discussed in earlier chapters for verifying, adding and recover/adapting internal and external constraints in web service composition environments. Then, we propose a *context/constraint-aware service brokerage* that includes the architecture of the system that can create and execute policy-based composite packages. Service brokerage is a mechanism that takes the role of inter-mediation among service requesters and service providers in different domains such as Service Oriented Architecture and Cloud computing. This inter-mediation role typically covers a broad range of responsibilities including service discovery and recommendation, service monitoring, Service- Level Agreement (SLA) management (among others), in many researches [67, 30, 5, 6]. In [5, 6], Badidi presents a cloud service broker framework for SaaS provisioning that is based on brokered SLA. The framework relies on a cloud service broker which is in charge of mediating between service consumers and SaaS providers, selecting appropriate SaaS providers, and negotiating the SLA terms. Moore and Mahmoud propose a trusted service broker for SaaS applications [66]. This broker acts as a repository proxy for the publication of heterogeneous SaaS applications from providers. The broker takes care of data integration issues that arise when there are data exchanges between different autonomous sources. As opposed to these other solutions, our proposed brokerage only aims at managing context/constraints in composite service creation and execution.

In this chapter, using the shopping scenario discussed in Chapter 5, we clarify the problem of applying different types of constraints on a constraint-aware composite service package. Then we update our algorithms from earlier chapters to make sure that when an external constraint is added to a policy-based package, the internal constraints of the services in the plan are verified properly in conjunction with the added external constraints. Finally, we propose a brokerage-based solution including all contributions to address all constraints-related issues discussed in earlier chapters. In our approach, we propose algorithms to generate and execute a policy- based

composite service package including alternative policy-based plans to execute users' tasks, adapt the package to new constraints and recover the composite service in face of failure of any constraint.

6.1 Motivation Scenario and Problem Analysis

In Section 5.1, we discussed a shopping scenario where each online store has its own specific external constraints, i.e. policies. In Table 11, we discussed three different policies to be added to the constraint-aware composite plan. Note that in none of those cases are the individual services of the constraint-aware composite service aware of the external constraint, nor are they responsible for it. Figure 23 depicts the policy-based composite service (based the services presented in Table 10) when:

$$C_{s1} = \{C_{31}, C_{41}\}, C_{s2} = \{C_{32}, C_{42}\}, C_{s3} = \emptyset, C_{s4} = \emptyset$$

$$P_1 = \emptyset, P_2 = \emptyset, P_3 = \emptyset, P_4 = \emptyset$$

.

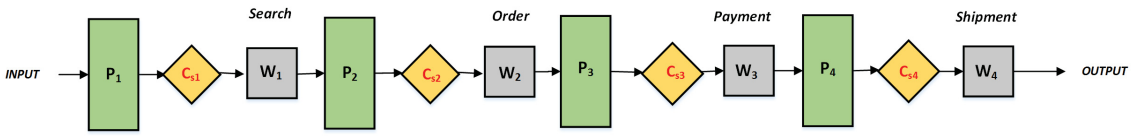


Figure 23: Shopping composite service

After all external constraints are added to the policy-based plan, the verification point of some internal constraints, which have been moved backward during constraint verification management, are not valid anymore and need to be readjusted again. For example, during the verification points adjustment, the verification point of $C_{32} = \{PaymentAmount \leq 10000\}$ changes to C_{s2} ($S_2 =< C_{s2}, W_2 >$) from C_{s3} (Figure 23). It is clear that verification of C_{32} depends on the value assigned to $PaymentAmount$ parameter. If we want to add p_3 to the plan, based on what we discussed in Chapter 5, it will be added to P_3. Now, imagine a user wants to purchase

a book, which costs 9000 ($PaymentAmount = 9000$), using this shopping service. During the shopping execution process, C_{32} is verified before execution of W_2 which is verified successfully as $PaymentAmount \leq 10000$. Then, after executing W_2, p_3 is applied which add 15% tax price to the total payment ($PaymentAmount = 1135$). In this situation, W_3 cannot be executed as $PaymentAmount \geq 10000$, which fails the execution of whole composite service. The reason is that applying p_3 can change the value assigned to $paymentAmount$ as $p_3.E = paymentAmount$. Therefore, after p_3 is added to the plan, the verification point of C_{32} is not valid anymore because it can affect the verification of C_{32} during the execution of the composite service. The solution to this problem is to move the verification points of C_{32} back to C_{s3}.

After policies are added to the plan, internal constraints of the constraint-aware composite service are verified inside the composite plan as follow:

$$C_{s1} = \{C_{31}, C_{41}\}, C_{s2} = \{C_{42}\}, C_{s3} = \{C_{32}\}, C_{s4} = \emptyset$$

$$P_1 = \emptyset, P_2 = \emptyset, P_3 = \{p_1, p_3\}, P_4 = \{p_2\}$$

.

The constraint management verification points for internal services in a constraint -aware composite plan might change as new external constraints will be added or removed from the plan. As a result, our internal constraint verification management which discussed in Chapter 3 needs to be updated to dynamically change verification point of internal constraints inside a policy-based composite service.

6.2 Policy-based Composite Package

Considering the issue relates to adding policies to a constraint-aware plans, in this section we propose our novel structure including alternative constraint-aware composite plans. The creation of the package is not different from the CaCSP which we discussed in Chapter 4. However, the policy adaptation algorithm from Chapter 5 needs to be modified to re-adjust the verification points of services‘ constraints inside

a package according to the effects associated with the injected policies. Finally, an approach is discussed to apply effects of policies during execution of a policy-based composite package.

6.2.1 Policy-based Composite Package Creation

In this section, we develop a new algorithm to make sure that after adding policies to a package, the verification points of internal constraints are appropriately adjusted. First, we need to define new concepts.

Definition 15. *A **Policy-based composite package** is a constraint-aware service composition package including a set of policy-based constraint-aware plans that can accomplish the same task. Each node in a policy-based package is a tuple* $< P, s >$ *where* $s =< C_S, service >$ *is a service-node and* P *is the set of all policies that are to be applied before* s *in the package.*

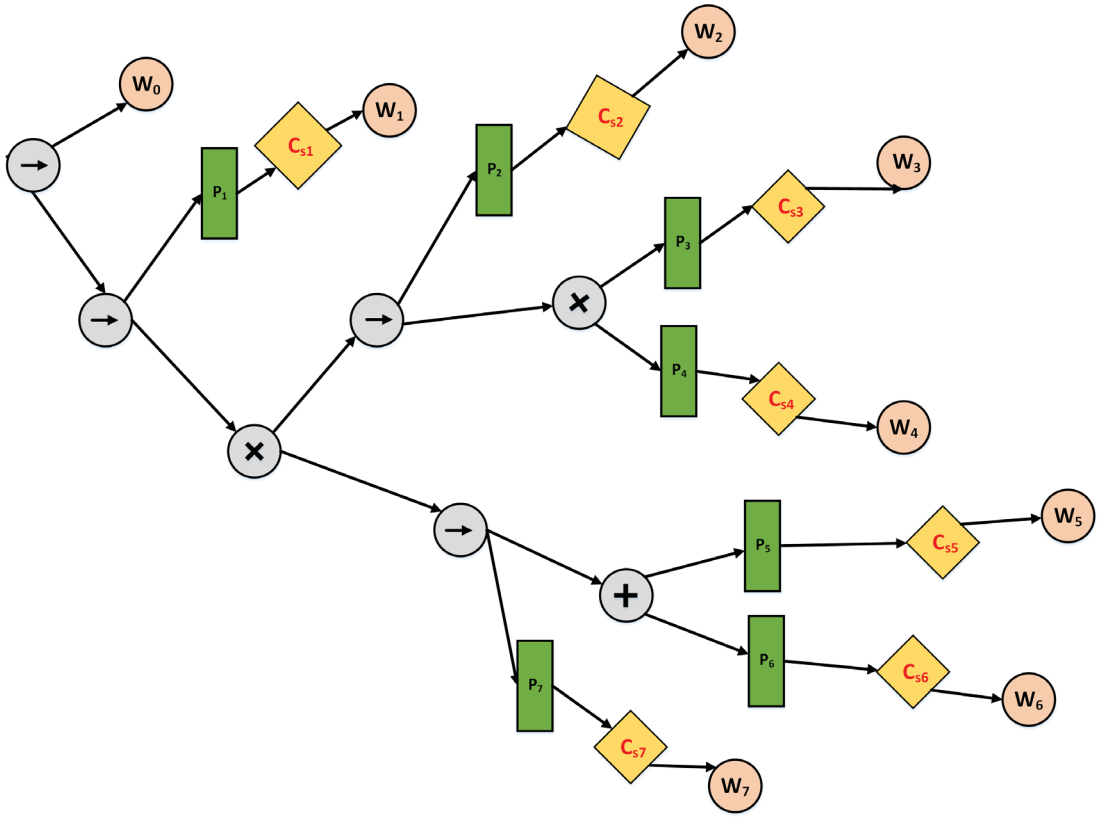


Figure 24: Complete package including internal and external constraints

Figure 24 shows the policy-based package of the scenario discussed in Section 3.1. Note that the specifications and constraints of services which are depicted in this figure are presented in Table 2.

In Chapter 5, Algorithm 8 discusses our policy-based adaptation process for a policy-based composite plan. Adding a policy to a policy-based composite package uses the same technique as adding a policy to a policy-based composite plan (Chapter 5), except anytime a policy is added to the package *adjustconstriant* (Algorithm 9) readjusts the verification points of service constraints to make sure they will be verified properly (Section 6.1), taking into consideration the potential effects of the injected policy. To add a policy to a policy-based composite package, the proper position of the policy inside the package needs to be found. This position is dependent on the data model of services inside the package. A policy ($p =< C_p, E_p >$) needs to be applied before execution of a service-node in a package, if the data model accumulated up to its execution includes all parameters in E_p. Then if the policy can be added before a service, all required parameter values to evaluate the policy constraints need to be available at runtime.

Algorithm 9 implements our mechanism to re-adjust internal constraints verification points inside a composite plan after the injection of a policy. Based on what we discussed in Section 6.1, anytime a policy (*policy*) is added before a service (*service*) inside a policy-based package, the verification points of all services' constraints of successors services, which have been moved before the *service* and *policy* affects their verification, need to be moved forward (line 8-15). To do that, for every internal constraint (*constraint*) of predecessor services, $belongsTo(constraint, succServices)$ checks to see whether the constraint belongs to any of successor services of *service*. It also checks to see whether the verification of the constraint is affected by applying the policy (Line 10). In this case, the constraint verification point are moved the service node which will be executed right after applying the policy (line 12).

Now, if we apply policies in Table 11 to the policy-based composite service package in Figure 24, Table 13 and Table 14 show how policies are added and internal

Table 13: Constraints verification plan after applying policies

	C_{s1}	C_{s2}	C_{s3}	C_{s4}	C_{s5}	C_{s6}	C_{s7}
Constraints	c_1, c_2, c_{31} c_{41}, c_{71} c_5, c_6	c_{32}, c_{42}, c_{72}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Table 14: Policies in the policy-based composite service package

	P_1	P_2	P_3	P_4	P_5	P_6	P_7
Policies		p_1, p_3	p_2	p_2		p_1, p_3	p_2

constraints are going to be verified in the package. As it is clear from Table 13, even after applying policies, the verification point of none of the policies will be moved back.

6.2.2 Policy-based Composite Package Execution

The execution of a policy-based package is similar to execution of a CaCSP, as defined earlier. However, before the execution of each service-node, the constraint of all policies that have been added to the package need to be applied based on the execution context. Then, if the policies are triggered, their effects need to be applied to the context. To apply effects of a policy, there are two points that need to be considered. First, in the dynamic environment of the web, policies can be applied and removed at anytime. As the service composition environment is very dynamic and it changes constantly, the set of policies that are added to a composite package could change dynamically and it potentially takes considerable resources to update policy-based packages based on new added policies. In case the added policy is not valid anymore the execution algorithm skips the policy and removes it from the policy-based composite package. Second, if all constraints in the set of constraints of the policy is verified to true, i.e. it becomes applicable, the effects of the policy will be applied to execution context (state) of the composite service.

Definition 16. *A policy ($p =< C_P, E_P >$) which has been added to a package is* ***applicable*** *to a state (context) $S = \{< T_1 \mapsto v_1 > ... < T_n \mapsto v_n >\}$ (where $\{T_1, T_2, ..., T_n\}$ is a set of ontology types representing all features in the data model*

Algorithm 9 adjustConstraints

Input: *policy_pkg* (a policy-based package), *service* (a service that the policy is added before that in the plan)

Output: *policy_pkg* (a policy-based package added with the new policy, if applicable)

```
1: preSrv = all predecessors of service in the package plan
2: sucSrv = all successors of service in the package plan
3: l = layer of service in plan_pkg
4: repeat
5:    l = l - 1
6:    services = all services in layer l of plan_pkg
7:    srvSet = services ∩ preSrv
8:    for (each srv ∈ srvSet) do
9:       for (each constraint ∈ srv.C) do
10:         if (belongsTO(constraint, succServices) and (constriant.E == policy.E)
            then
11:            srv.C = srv.C − constraint
12:            service.C = service.C ∪ constraint
13:         end if
14:      end for
15:   end for
16: until (preServ = ∅)and(l >= 0)
17: return policy_pkg
```

of the package, and $\{v_1, ...v_n\}$ *are literal values of the same respective types), if verification of all constraints in* C_P *would be satisfied based on the values assigned to the parameters in* S*.*

In our model, during the execution of a policy-based composite package, when a policy ($p =< C_P, E_P >$) is applicable to state S, a **Policy State Transition Function** (γ_p) changes the state of execution to S' : $S' = \gamma_p(S, P)$. We define **Policy State Transition Function** as follow:

Definition 17. ***Policy State Transition Function*** *(γ_p) is a function that applies the effects of a policy(p) on the state (S) of the composite service, if the policy is applicable.*

Algorithm 10 presents our policy package execution algorithm. This algorithm is very similar to Algorithm 7, except that before the execution of each service-node in the package, it checks for all applicable policies and applies their effects on the execution context of the package (line 4-8).

To avoid any confusion, it should be noted that when a policy is added/applied to a policy-based composite package, it is injected inside the package. However, when a policy is applicable to a state (context) at execution time, it applies its effects by modifying the values assigned to state parameters based on the policy's effects (E).

6.3 Constraint-aware Web Service Brokerage

In this section, we discuss the architecture and information model of our proposed *context/constraint-aware service brokerage* that includes all discussed contributions presented in earlier chapters to manage different aspects related to context and constraints in web service composition environment. A *service brokerage* is defined as a mechanism for the problem of *service provisioning*, which takes the role of intermediation among clients looking for a service and providers offering a service [43]. In our approach, we propose a *context/constraint-aware service brokerage* as a solution to enable the connection among service requester, service providers, service users and

Algorithm 10 policy-package_execution

Input: *policy_pkg* (a policy-based package), S_0 (initial state of execution)
Output: either execution state state or NULL

1: $result = Null$
2: **if** $(IsPolicyNode(policy\_pkg)$ **then**
3: $policy\_node = policy\_pkg$
4: **for** (each $policy \in policy\_node.P$) **do**
5: **if** $applicable(policy)$ **then**
6: $StateList[pkg\_plan] = \gamma_p(S_0, policy)$
7: **end if**
8: **end for**
9: **if** $(GSD(policy\_node.S.C_S), S_0)$ **then**
10: $StateList[pkg\_plan] = \gamma(S_0, policy\_node.S.W)$
11: **return** $StateList[pkg\_plan]$
12: **else**
13: $Prune(pkg\_plan)$
14: **return** $result = cmp\_pkg\_execution(pkg\_plan, S_0)$
15: **end if**
16: **else**
17: $operator$ = The operator in the root of the composite package
18: $t_1 = leftSubTree(policy\_pkg, operator)$
19: $t_2 = rightSubTree(policy\_pkg, operator)$
20: **if** $(operator\ is\ \rightarrow)$ **then**
21: $result = policy - package\_execution(t_1, S_0)$
22: $result = policy - package\_execution(t_2, result)$
23: **end if**
24: **if** $(operator\ is\ \otimes)$ **then**
25: $result = policy - package\_execution(t_1, S_0)$
26: **if** $(result\ is\ Null)$ **then**
27: $result = policy - package\_execution(t_2, S_0)$
28: **end if**
29: **end if**
30: **if** $(operator\ is\ \oplus)$ **then**
31: $result = policy - package\_execution(t_1, S_0)$
32: $result = policy - package\_execution(t_2, result)$
33: **end if**
34: **end if**
35: **return** $result$

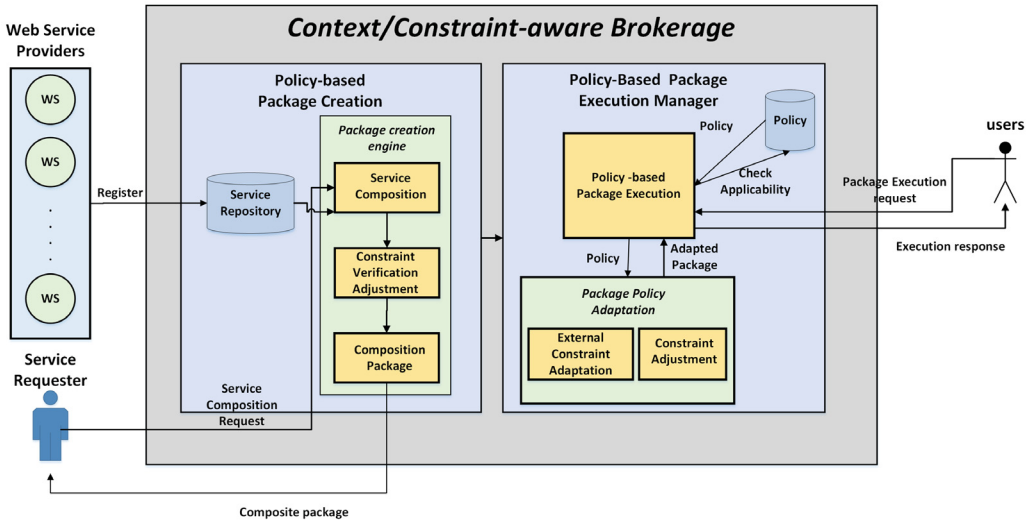


Figure 25: Architecture of context/constraint-aware web service brokerage

execution environment and generate and execute context-aware constraint/policy-based packages.

6.3.1 Architecture

In this section, we present an abstract architecture of our context/constraint- aware web service brokerage to perform package creation, recovery and adaptation in a service environment. The architecture presents the general perspective of our approach. However, it should be mentioned that it is not fully implemented. Each sub-component represents an algorithm that we have presented in last four chapters. Figure 25 depicts the architecture including the two main components: *policy-based package creation* and *policy-based package execution manager*.

The policy-based package creation component designs the policy-based composite packages while taking into consideration requesters and providers constraints. The process starts with a composite service request which is made by a service requester to the service composition component. The *Service Composition* component generates a set of alternative constraint- aware composite services based on our graph plan based algorithms discussed in Section 3.4. The output of this component, as it is discussed in Chapter 3, is a set of alternative policy-based plans that can accomplish the

required task in web service composition request. Then, the *Constraint Verification Adjustment* component adjusts verification points of internal constraints inside each constraint- aware composite service (Algorithm 5). The output for this component is the set of alternative plans with adjusted internal service constraints. Finally, the *Package Composition* component creates a constraint-aware composite package according to Algorithm 6. After the composite package is created, it is provided for execution to the service requester. In addition, anytime a service user executes the package, the *Policy-Based Package Execution Manager* manages execution of the package. The Policy-Based Package Execution Manager component has two main sub components, namely *Policy-Based Package Execution* and *Constraint Adjustment*. The Policy-based Package Execution component executes policy-based packages based on Algorithm 10. In addition, in case of failure it switches among different policy-based composite plans. In addition, in case a new policy will be added, the *Package Policy Adaptation* component adds new policies to the package (Algorithm 8) and adjusts internal constraints (Algorithm 9) accordingly inside the package.

6.3.2 Information Model

Figure 26 shows a high-level overview of the proposed information model of context/constraint-aware service brokerage. Our information model revolves around modeling two central concepts and their relationships: *Service* and *Constraint*.

A *service requester* can make 1 to many *service requests* and for each *service request*, it could have 0 to many possible *services/composite services* which are also provided by *service providers*. Each *service-node* is composed of a *services*, which can have 0 to many *constraints*. In addition, each *service* has a *data model* which also composes the *state* of the *service*. Each *policy-based constraint-aware plan* is composed of 1 to many *service-nodes* and 0 to many *policies*. Each *composite package* is also composed of 1 to many *policy-based constraint-aware plans*.

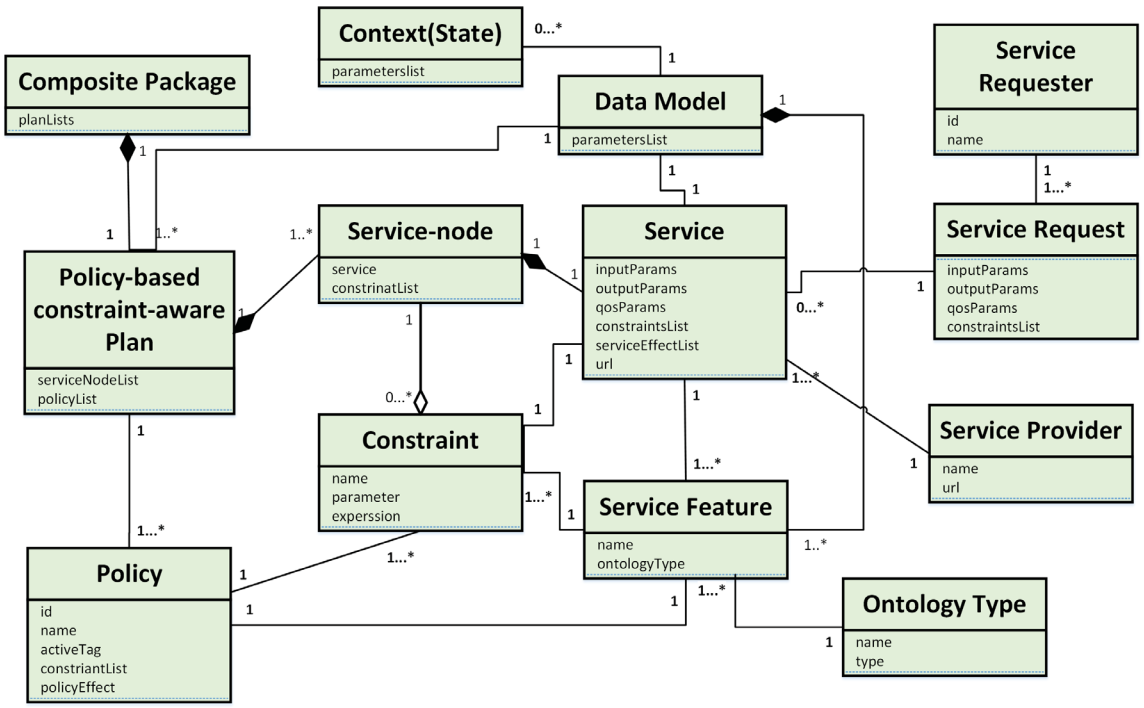


Figure 26: Information model of context/constraint-aware service brokerage

6.4 Evaluation, Discussions, and Summary

In this chapter, we have proposed a policy-based composite service package including all discussed contributions in earlier chapters to verify and apply different types of constraints at runtime. We also proposed an architecture for a context/constraint-aware brokerage to discuss different components of a system that can create and execute policy-based composite packages. In addition, we also discussed that adding policies can interfere with the verification of internal constraints during the execution of composite services. To address this issue, we proposed a new algorithm (Algorithm 9) and updated the proposed algorithms from Chapter 4 and Chapter 5 to deal with this issue.

In addition, it is very important to note that finding all alternative composite solutions for the web service composition problem is a well- known NP-complete problem. To create a policy-based composite package, we do not need to have all possible solution plans, and only having a subset of all solution plans is enough. Therefore, in order to avoid this fundamental problem for the brokerage,

the number of solution plans could be bounded, where we let the composition algorithm (Algorithm 1) stop after it has found a fixed number of solutions. To make sure there is at least one possible plan for a web service composition problem, there are approaches like [102] that can find a solution in polynomial time, but with possible redundant services. Therefore, in case the algorithm could not find any solution after a certain time, those approaches can be used to find a possible solution for the problem in polynomial time. Then, a policy-based composite service package including at least one possible constraint-aware plan can be constructed. The complexity of Algorithm 9 is polynomial since the complexity of this algorithm depends on the number of composition plans, services and internal constraints inside the package. If the maximum number of constraints in each constraint set is β and the maximum number of services in one composition plan is n, which is the number of services in the repository. In the best case, no constraint can be moved back after applying a new policy to the package, which makes the complexity $O(mn\beta)$. However, in the worst case, all constraints of all services of all composition plans could be moved back to the position after policy injection point which makes the complexity $(O(mn^2\beta))$. Algorithm 10 is also a recursive algorithm in which the number of recursions depends on the number of services in the package. If the number alternative plans is limited to m and the maximum number of services in each plan is n the complexity of this algorithm is $O(nm)$ which is polynomial.

We also discussed the architecture and the high level information model of our policy-based composite service brokerage in this chapter. The architecture is designed such that every component refers to a set of algorithms as we defined earlier. Therefore, in case new algorithms are developed in the feature, they can be replaced by specific components. In designing a policy-based composition package, a service can appear at multiple locations in a policy-based composite service package, but its executions does not necessarily have the same results as its execution in other branches. It is also possible to have multiple permutations for a constraint-aware composite plan (service nodes sequences) in a package. However, it should be noted that different permutations do not necessarily have the same execution results and

their execution result cannot be substituted. In addition, the execution of a task by a branch of the package could be failed because of failure in verification of service constraint. However, another branch can complete the task and satisfy the verification of the same constraint, the constraint being evaluated in a different context. In addition, in applying different policies to a policy-based composite package, the order in which the policies are applied might affect the execution result. However, as this thesis we made the assumption that any number of policies could be added to a package and that applying all policies in any order have the same results. In this way, we do not have to consider the priority of applying different policies in a policy-based composite package.

Chapter 7

Conclusions and Future Work

Although web service composition has received considerable attention in different domains and it is seen as a promising way to create services for more complex tasks, it still raises many challenges. One of the challenges relates to considering different types of constraints in design and execution of composite services. As services in the real world are not universally applicable, they cannot be executed correctly in all contexts. Services have applicable conditions and usage restrictions that require being considered during the composition process, as well as during the execution of composite services. In addition to service constraints, some other limitations (external constraints) might be imposed on working composite systems which are not service constraints. In this chapter, we conclude with a summary of the contributions of this thesis and discuss the planned future work.

7.1 Discussion

We have encountered many challenges in verification of service constraints and external constraints during execution of composite services. There are only a few approaches that consider constraint verification in web service composition execution. We realized failure in verification of constrains of services inside the composite plan can fail execution of the composite service and result in wasting of time and computational resources. Constraint verification may result in composite service

plan failure if some of the constraints are not satisfied at runtime. Thus, the failed composition plan needs to be recovered to complete a failed execution. Current verification and failure recovery approaches are wasting computational resources as they only consider input/output parameters and do not consider constraints in web service composition and execution. We realized that verification of service constraints inside a composite service depends on the execution results of other services inside the plan. Therefore, we modeled these dependencies and improve constraint verification and failure recovery process in a web service composition to reduce the number of rollback penalties and wastage of computational resources. Another primary challenge is to apply effects of external constraints on working composite services. External constraints can be applied during execution of composite services. In this situation, although all services in a composite service may perform well, the execution results of the composite service might not be valid based on the emerging external constraints. We realized that using current adaptation approaches to adapt a working composite service according to external constraints is not always successful and can add considerable overhead to the system.

7.2 Summary of Contributions

Considering the discussed challenges, here are the contributions of our thesis which have been discussed in details in last 6 chapters. First, a model and constraint-aware service composition algorithms has been developed to have a clear and sound understanding of all related concepts. We developed a model for web service composition to formally express all essential concepts such as service, constraint, data model, and context. We also proposed algorithms based on planning-graph to generate constraint-aware composite services, where constraint-aware composite services are directed graphs that verify internal constraints of component services in a plan before their execution.

In the second contribution, a novel constraint verification approach has been

designed to adjust the verification points of service constraints inside a constraint-aware composite service. This approach has been demonstrated to reduce the cost of possible rollbacks which are necessitated by the constraint verification failure of individual services. This approach models dependencies among services inside a composite service. Then, it moves back the verification points of service constraints inside a composition plan to avoid unnecessary execution of services. As our evaluation proved, using our solution, the number of unnecessary execution of component services could be reduced around 50%. In addition, using the proposed verification method, an upcoming failure during the execution of composite service can be caught faster. A failure recovery approach is proposed to start recovery as soon as an upcoming constraint verification failure is caught during execution of composite services. Further along, in our proposed approach, we focused on defining a novel structure, which we called constraint-aware composite package, which includes different constraint-aware service composition solutions to recover failure at execution time. The constraint-aware composite package helps to have a better perspective through execution plans at runtime and considering service constraints recover the failure as soon as possible.

In our next contribution, external constraint is represented and expressed as policies based on the definitions of our proposed model. Then, an adaptation approach for externally defined policies/constraints in web service composition was proposed. External constraints are formally defined and a solution to adapt a composite service to external constraints is provided. The proposed approach can add/remove external constraints inside a composite service without re-construction of the plan. Therefore, compared to other adaptation approaches, the runtime performance of adaptable composite web services is significantly improved compared to existing solutions. Finally, we proposed Policy-based Composite Package which is a novel structure including alternative policy-based plans for a web service composite request. It can handle all approaches discussed in earlier chapters for verifying and recovering internal constraints, and applying and adding external constraints. In addition, an architecture of a context/constraint-aware service brokerage to represent the behavior

of our proposed system to manage creation, execution, and adaptation of policy-based packages is presented.

7.3 Future Work

In our future work, we plan to improve the scalability of our context/constraint-aware service brokerage. If the brokerage receives many composition requests, faces many verification failures inside composite services, and has to adapt many composite services to many policies, it should perform as well as other adaptation approaches. Therefore, it could be interesting to test the effectiveness of the brokerage and improve the scalability of the brokerage. We plan to use GIPSY [75] which is a distributed computational system, to test the scalability of our work in a simulated environment. Therefore, composite services should be expressed using the Lucid dataflow programming language [72].

In our approach, alternative policy-based plans can be composed in different ways in a policy-based composite service package. One future work could be defining specific quality features for the package and create more efficient packages considering those quality features. For example packages that have less response time compared to other packages that can be developed for a web service composition request. One solution is to use Genetic Algorithms to improve the quality of proposed packages to have fewer rollback penalties at runtime compared to other packages.

Finally, it is interesting to use our proposed approach in different environments where services in composite plans have more internal constraints and during their execution other external constraints affect execution of composite services. IoT (Internet of things) is such an environment, in which services which represent different devices, have different types of constraints. In addition, failure recovery and adaptation of composite services in IoT has become more and more interesting since IoT services have different constraints that need to be considered during the design and execution of IoT composite services. We believe that our approaches can help to develop a more robust and reliable service provisioning system for the IoT

environment. In this way, services can be composed considering their constraints and in case of failure or new environment context, they can be recovered and adapted.

Bibliography

[1] Alférez, G., Pelechano, V., Mazo, R., Salinesi, C., and Diaz, D. (2014). Dynamic adaptation of service compositions with variability models. *Journal of Systems and Software*, 91:24–47.

[2] Alrifai, M. and Risse, T. (2009). Combining global optimization with local selection for efficient qos-aware service composition. In *Proceedings of the 18th international conference on World wide web*, pages 881–890. ACM.

[3] Alrifai, M., Skoutas, D., and Risse, T. (2010). Selecting skyline services for qos-based web service composition. In *Proceedings of the 19th international conference on World wide web*, pages 11–20. ACM.

[4] Aznag, M., Quafafou, M., and Jarir, Z. (2014). Leveraging formal concept analysis with topic correlation for service clustering and discovery. In *Web Services (ICWS), 2014 IEEE International Conference on*, pages 153–160. IEEE.

[5] Badidi, E. (2013a). A cloud service broker for SLA-based SaaS provisioning. In *Proceedings of the 2013 International Conference on Information Society (i-Society)*, pages 61–66.

[6] Badidi, E. (2013b). A framework for Software-as-a-Service selection and provisioning. *CoRR*, abs/1306.1888. `http://arxiv.org/abs/1306.1888`.

[7] Baldauf, M., Dustdar, S., and Rosenberg, F. (2007). A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277.

[8] Baresi, L. and Guinea, S. (2011). Self-supervising BPEL processes. *IEEE Transactions on Software Engineering*, 37(2):247–263.

[9] Bentaleb, A. and Ettalbi, A. (2017). Context-aware for service composition optimization in cloud computing. In *International Conference on Information Technology and Communication Systems*, pages 311–321. Springer.

[10] Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., and Mecella, M. (2003). Automatic composition of e-services that export their behavior. In *Service-Oriented Computing-ICSOC 2003*, pages 43–58. Springer.

[11] Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., and Mecella, M. (2005). Automatic service composition based on behavioral descriptions. *International Journal of Cooperative Information Systems*, 14(04):333–376.

[12] Berbner, R., Spahn, M., Repp, N., Heckmann, O., and Steinmetz, R. (2006). Heuristics for qos-aware web service composition. In *Web Services, 2006. ICWS'06. International Conference on*, pages 72–82. IEEE.

[13] Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A., and Riboni, D. (2010). A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 6(2):161–180.

[14] Bleul, S., Weise, T., and Geihs, K. (2009). The web service challenge-a review on semantic web service composition. volume 17.

[15] Blum, A. L. and Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial intelligence*, 90(1):281–300.

[16] Boella, G. and Damiano, R. (2002). A replanning algorithm for a reactive agent architecture. *Artificial Intelligence: Methodology, Systems, and Applications*, pages 359–387.

[17] Brogi, A. and Corfini, S. (2007). Behaviour-aware discovery of web service compositions. *International Journal of Web Services Research*, 4(3):1.

[18] Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., and Tamburrelli, G. (2011). Dynamic QoS management and optimization in service-based systems. *IEEE Transactions on Software Engineering*, 37(3):387–409.

[19] Canfora, G., Di Penta, M., Esposito, R., and Villani, M. L. (2005a). An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1069–1075. ACM.

[20] Canfora, G., Di Penta, M., Esposito, R., and Villani, M. L. (2005b). Qos-aware replanning of composite web services. In *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*, pages 121–129. IEEE.

[21] Cavallaro, L., Di Nitto, E., and Pradella, M. (2009). An automatic approach to enable replacement of conversational services. In *Service-Oriented Computing*, pages 159–174. Springer.

[22] Chafle, G., Doshi, P., Harney, J., Mittal, S., and Srivastava, B. (2007). Improved adaptation of web service compositions using value of changed information. pages 784–791.

[23] Chen, I. Y., Yang, S. J., and Zhang, J. (2006). Ubiquitous provision of context aware web services. In *Services Computing, 2006. SCC'06. IEEE International Conference on*, pages 60–68. IEEE.

[24] Chen, M. and Yan, Y. (2012). Redundant service removal in qos-aware service composition. In *Web Services (ICWS), 2012 IEEE 19th International Conference on*, pages 431–439. IEEE.

[25] Colombo, M., Nitto, E. D., and Mauri, M. (2006). SCENE: A service composition execution environment supporting dynamic changes disciplined through rules. 4294.

[26] da Silva, A. S., Ma, H., and Zhang, M. (2014). A graph-based particle swarm optimisation approach to qos-aware web service composition and selection. In

Evolutionary Computation (CEC), 2014 IEEE Congress on, pages 3127–3134. IEEE.

[27] Dolog, P., Schäfer, M., and Nejdl, W. (2014). Design and management of web service transactions with forward recovery. In *Advanced Web Services*, pages 3–27. Springer.

[28] Dustdar, S. and Schreiner, W. (2005). A survey on web services composition. *Int. J. Web Grid Serv.*

[29] El Hadad, J., Manouvrier, M., and Rukoz, M. (2010). Tqos: Transactional and qos-aware selection algorithm for automatic web service composition. *IEEE Transactions on Services Computing*, 3(1):73–85.

[30] Ferrer, A. J., HernáNdez, F., Tordsson, J., Elmroth, E., Ali-Eldin, A., Zsigri, C., Sirvent, R., Guitart, J., Badia, R. M., Djemame, K., et al. (2012). OPTIMIS: A holistic approach to cloud service provisioning. *Future Generation Computer Systems*, 28(1):66–77.

[31] Fox, M., Gerevini, A., Long, D., and Serina, I. (2006). Plan stability: Replanning versus plan repair. In *ICAPS*, volume 6, pages 212–221.

[32] Gao, L., Urban, S. D., and Ramachandran, J. (2011). A survey of transactional issues for web service composition and recovery. *International Journal of Web and Grid Services*, 7(4):331–356.

[33] Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated Planning: theory and practice*. Elsevier.

[34] Grigori, D., Corrales, J. C., and Bouzeghoub, M. (2006). Behavioral matchmaking for service retrieval. In *ICWS'06*, pages 145–152. IEEE.

[35] Hamadi, R. and Benatallah, B. (2003). A petri net-based model for web service composition. In *Proceedings of the 14th Australasian database conference-Volume 17*, pages 191–200. Australian Computer Society, Inc.

[36] Han, B., Jia, W., Shen, J., and Yuen, M.-C. (2004). Context-awareness in mobile web services. In *International Symposium on Parallel and Distributed Processing and Applications*, pages 519–528. Springer.

[37] Hashemian, S. V. and Mavaddat, F. (2005). A graph-based approach to web services composition. In *The 2005 Symposium on Applications and the Internet*, pages 183–189.

[38] Hassine, A. B., Matsubara, S., and Ishida, T. (2006). A constraint-based approach to horizontal web service composition. In *ISWC 2006*, pages 130–143. Springer.

[39] Henricksen, K., Indulska, J., and Rakotonirainy, A. (2002). Modeling context information in pervasive computing systems. In *Pervasive Computing*, pages 167–180. Springer.

[40] Koning, M., a. Sun, C., Sinnema, M., and Avgeriou, P. (2009). VxBPEL: Supporting variability for web services in BPEL. *Information and Software Technology*, 51(2).

[41] Kwon, J. and Lee, D. (2012). Non-redundant web services composition based on a two-phase algorithm. *Data & Knowledge Engineering*, 71(1):69–91.

[42] Laleh, T., Khodadadi, A., Mokhov, S. A., Paquet, J., and Yan, Y. (2014). Toward policy-based dynamic context-aware adaptation architecture for web service composition. In *Proceedings of C3S2E'14*, pages 158–163. Short paper.

[43] Laleh, T., Mokhov, S. A., Paquet, J., and Yan, Y. (2015). Context-aware cloud service brokerage: A solution to the problem of data integration among SaaS providers. In Desai, B. C. and Toyoma, M., editors, *Proceedings of the Eighth International C* Conference on Computer Science & Software Engineering*, C3S2E 2015, pages 46–55.

[44] Laleh, T., Paquet, J., Mokhov, S., and Yan, Y. (2017). Predictive failure recovery

in constraint-aware web service composition. In *Proceedings of the 7th International Conference on Cloud Computing and Services Science*, pages 241–252.

[45] Laleh, T., Paquet, J., Mokhov, S. A., and Yan, Y. (2016a). Efficient constraint verification in service composition design and execution (short paper). In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 445–455. Springer.

[46] Laleh, T., Paquet, J., Mokhov, S. A., and Yan, Y. (2016b). Efficient constraint verification in service composition design and execution (short paper). In *CoopIS*, pages 445–455. Springer.

[47] Lécué, F. and Léger, A. (2006). A formal model for semantic web service composition. In *The Semantic Web-ISWC 2006*, pages 385–398. Springer.

[48] Lemos, A. L., Daniel, F., and Benatallah, B. (2015). Web service composition: A survey of techniques and tools. *ACM Computing Surveys (CSUR)*, 48(3):33.

[49] Li, J. (2016). *Full Solution Indexing and Efficient Compressed Graph Representation for Web Service Composition*. PhD thesis, Concordia University.

[50] Li, J., Yan, Y., and Lemire, D. Full solution indexing for top-k web service composition.

[51] Liang, Q. A. and Su, S. Y. (2005). And/or graph and search algorithm for discovering composite web services. *International Journal of Web Services Research*, 2(4):48.

[52] Lin, C.-F., Sheu, R.-K., Chang, Y.-S., and Yuan, S.-M. (2011). A relaxable service selection algorithm for qos-based web service composition. *Information and Software Technology*, 53(12):1370–1381.

[53] Lin, K.-J., Zhang, J., and Zhai, Y. (2009). An efficient approach for service process reconfiguration in soa with end-to-end qos constraints. In *Commerce and Enterprise Computing, 2009. CEC'09. IEEE Conference on*, pages 146–153. IEEE.

[54] Lin, K.-J., Zhang, J., Zhai, Y., and Xu, B. (2010). The design and implementation of service process reconfiguration with end-to-end qos constraints in soa. *Service Oriented Computing and Applications*, 4(3):157–168.

[55] Mabrouk, N. B., Beauche, S., Kuznetsova, E., Georgantas, N., and Issarny, V. (2009). Qos-aware service composition in dynamic service oriented environments. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, page 7. Springer-Verlag New York, Inc.

[56] Manes, A. T. (2001). Enabling open, interoperable, and smart web services the need for shared context. In *Proc W3C Web Services Workshop*.

[57] Marconi, A. and Pistore, M. (2009). Synthesis and composition of web services. In Bernardo, M., Padovani, L., and Zavattaro, G., editors, *Formal Methods for Web Services*, volume 5569 of *Lecture Notes in Computer Science*, pages 89–157. Springer Berlin Heidelberg.

[58] McCarthy, J. (1993). Notes on formalizing context.

[59] McCarthy, J. and Buvac, S. (1997). Formalizing context (expanded notes).

[60] McIlraith, S. and Son, T. C. (2002). Adapting golog for composition of semantic web services. *KR*, 2:482–493.

[61] Medjahed, B. and Atif, Y. (2007). Context-based matching for web service composition.

[62] Menasce, D., Gomaa, H., Malek, S., and Sousa, J. P. (2011). SASSY: A framework for self-architecting service-oriented systems. *IEEE Software*, 28(6):78–85.

[63] Meyer, H., Kuropka, D., and Tröger, P. (2007). Asg-techniques of adaptivity. In *Autonomous and Adaptive Web Services*.

[64] Meyer, H. and Weske, M. (2006). Automated service composition using heuristic search. In Dustdar, S., Fiadeiro, J., and Sheth, A. P., editors, *Business Process*

Management, volume 4102 of *Lecture Notes in Computer Science*, pages 81–96. Springer Berlin Heidelberg.

[65] Microsoft (2001). Global weather web service. `http://www.webservicex.com/globalweather.asmx?WSDL`.

[66] Moore, B. and Mahmoud, Q. H. (2009). A service broker and business model for SaaS applications. In *Proceedings of the IEEE/ACS International Conference on Computer Systems and Applications (AICCSA 2009)*, pages 322–329. IEEE.

[67] Moscato, F., Aversa, R., Di Martino, B., Fortis, T., and Munteanu, V. (2011). An analysis of mOSAIC ontology for cloud resources annotation. In *Proceedings of the 2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 973–980.

[68] Oh, S.-C., Lee, D., and Kumara, S. R. (2006). A comparative illustration of ai planning-based web services composition. *ACM SIGecom Exchanges*, 5(5):1–10.

[69] Oh, S.-C., Lee, D., and Kumara, S. R. (2008). Effective web service composition in diverse and large-scale service networks. *Services Computing, IEEE Transactions on*, 1(1):15–32.

[70] Oh, S.-C., Lee, D., and Kumara, S. R. T. (2007). Web service planner (wspr): An effective and scalable web service composition algorithm. *Int. J. Web Service Res.*, 4(1):1–22.

[71] Oh, S.-C., On, B.-W., Larson, E. J., and Lee, D. (2005). Bf*: Web services discovery and composition as graph search problem. In *e-Technology, e-Commerce and e-Service, 2005. EEE'05. Proceedings. The 2005 IEEE International Conference on*, pages 784–786. IEEE.

[72] Orchard, D. A. and Matthews, S. (2008). Integrating lucid's declarative dataflow paradigm into object-orientation. *Mathematics in Computer Science*, 2(1):103–122.

[73] Osland, P., Viken, B., Solsvik, F., Nygreen, G., Wedvik, J., and Myklbust, S. (2006). Enabling context-aware applications. *Proceedings of ICIN2006: Convergence in Services, Media and Networks*.

[74] Papazoglou, M. (2011). *Web services: principles and technology*. Pearson Education.

[75] Paquet, J. (2009). Distributed eductive execution of hybrid intensional programs. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*, pages 218–224. IEEE Computer Society.

[76] Peer, J. (2005). Web service composition as AI planning, a survey. [online]. Second revised version, `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.85.9119`.

[77] Perera, C., Zaslavsky, A., Christen, P., and Georgakopoulos, D. (2013). Context aware computing for the internet of things: A survey. *IEEE Communications Surveys & Tutorials Journal*, 2013. `http://arxiv.org/abs/1305.0982`.

[78] Ponnekanti, S. R. and Fox, A. (2002). Sword: A developer toolkit for web service composition. In *Proc. of the Eleventh International World Wide Web Conference, Honolulu, HI*, volume 45.

[79] Rao, J. and Su, X. (2005). A survey of automated web service composition methods. In Cardoso, J. and Sheth, A., editors, *Semantic Web Services and Web Process Composition*, volume 3387, pages 43–54. Springer Berlin Heidelberg.

[80] Sonntag, M. and Karastoyanova, D. (2011). Compensation of adapted service orchestration logic in bpel'n'aspects. In *Proceedings of the 9th International Conference on Business Process Management (BPM 2011)*, pages 1–16.

[81] Srivastava, U., Munagala, K., Widom, J., and Motwani, R. (2006). Query optimization over web services. In *Proceedings of the 32nd international conference on Very large data bases*, pages 355–366. VLDB Endowment.

[82] Strang, T. and Linnhoff-Popien, C. (2004). A context modeling survey. In *Workshop Proceedings*.

[83] Strunk, A. (2010). Qos-aware service composition: A survey. In *Web Services (ECOWS), 2010 IEEE 8th European Conference on*, pages 67–74. IEEE.

[84] Sun, W., Zhang, X., Yuan, Y., and Han, T. (2013). Context-aware web service composition framework based on agent. In *Information Technology and Applications (ITA), 2013 International Conference on*, pages 30–34. IEEE.

[85] Truong, H.-L. and Dustdar, S. (2009). A survey on context-aware web service systems. *International Journal of Web Information Systems*, 5(1):5–31.

[86] Van Der Krogt, R. and De Weerdt, M. (2005). Plan repair as an extension of planning. In *ICAPS*, volume 5, pages 161–170.

[87] Wagner, F., Ishikawa, F., and Honiden, S. (2011). Qos-aware automatic service composition by applying functional clustering. In *Web Services (ICWS), 2011 IEEE International Conference on*, pages 89–96. IEEE.

[88] Wang, B. and Tang, X. (2014). Designing a self-adaptive and context-aware service composition system. In *Computing, Communications and IT Applications Conference (ComComAp), 2014 IEEE*, pages 155–160. IEEE.

[89] Wang, H., Wang, X., Hu, X., Zhang, X., and Gu, M. (2016). A multi-agent reinforcement learning approach to dynamic service composition. *Information Sciences*, 363:96–119.

[90] Wang, H., Wu, Q., Chen, X., Yu, Q., Zheng, Z., and Bouguettaya, A. (2014a). Adaptive and dynamic service composition via multi-agent reinforcement learning. In *Web Services (ICWS), 2014 IEEE International Conference on*, pages 447–454. IEEE.

[91] Wang, H., Zhou, X., Zhou, X., Liu, W., and Li, W. (2010). Adaptive and dynamic service composition using q-learning. In *Tools with Artificial Intelligence*

(ICTAI), 2010 22nd IEEE International Conference on, volume 1, pages 145–152. IEEE.

[92] Wang, P., Ding, Z., Jiang, C., and Zhou, M. (2014b). Constraint-aware approach to web service composition. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 44(6):770–784.

[93] Wang, P., Ding, Z., Jiang, C., Zhou, M., and Zheng, Y. (2015). Automatic web service composition based on uncertainty execution effects.

[94] Wang, X. H., Zhang, D. Q., Gu, T., and Pung, H. K. (2004). Ontology based context modeling and reasoning using owl. In *Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on*, pages 18–22. Ieee.

[95] Wu, Q., Ishikawa, F., Zhu, Q., and Shin, D. H. (2016). Qos-aware multigranularity service composition: Modeling and optimization. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, PP(99):1–13.

[96] Xu, J., Li, Z., Chi, H., Wang, M., Guan, C., Reiff-Marganiec, S., and Shen, H. (2016). Optimized composite service transactions through execution results prediction. In *Web Services (ICWS), 2016 IEEE International Conference on*, pages 690–693. IEEE.

[97] Yan, Y., Chen, M., and Yang, Y. (2012a). Anytime qos optimization over the plangraph for web service composition. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1968–1975. ACM.

[98] Yan, Y., Chen, M., and Yang, Y. (2012b). Anytime qos optimization over the plangraph for web service composition. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1968–1975. ACM.

[99] Yan, Y., Poizat, P., and Zhao, L. (2010a). Repair vs. recomposition for broken service compositions. In *Service-Oriented Computing*, pages 152–166. Springer.

[100] Yan, Y., Poizat, P., and Zhao, L. (2010b). Repairing service compositions in a changing world. In Lee, R., Ormandjieva, O., Abran, A., and Constantinides, C., editors, *Proceedings of SERA 2010 (selected papers)*, volume 296 of *Studies in Computational Intelligence*, pages 17–36. Springer Berlin Heidelberg.

[101] Yan, Y., Poizat, P., and Zhao, L. (2010c). Self-adaptive service composition through graphplan repair. In *ICWS*, pages 624–627. IEEE.

[102] Yan, Y. and Zheng, X. (2008). A planning graph based algorithm for semantic web service composition. In *CEC/EEE 2008*, pages 339–342.

[103] Yu, T. and Lin, K.-J. (2005). Adaptive algorithms for finding replacement services in autonomic distributed business processes. In *Autonomous Decentralized Systems, 2005. ISADS 2005. Proceedings*, pages 427–434. IEEE.

[104] Yu, T., Zhang, Y., and Lin, K.-J. (2007). Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Transactions on the Web (TWEB)*, 1(1):6.

[105] Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., and Sheng, Q. Z. (2003). Quality driven web services composition. In *Proceedings of the 12th international conference on World Wide Web*, pages 411–421. ACM.

[106] Zhai, Y., Zhang, J., and Lin, K.-J. (2009). Soa middleware support for service process reconfiguration with end-to-end qos constraints. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 815–822. IEEE.

[107] Zheng, X. and Yan, Y. (2008). An efficient syntactic web service composition algorithm based on the planning graph model. In *ICWS'08*, pages 691–699. IEEE.

[108] Zhou, J., Gilman, E., Palola, J., Riekki, J., Ylianttila, M., and Sun, J. (2011). Context-aware pervasive service composition and its implementation. *Personal and Ubiquitous Computing*, 15(3):291–303.

Appendix

For experimental evaluations, we implemented the algorithms discussed in previous chapters. All experiments are performed on a PC platform with Intel CPU 3450 (2.67GHz), Windows 7, and 8GB RAM. The experimental platform is implemented in JAVA under the Eclipse environment. In addition, we use a platform which is initially developed for the Web Service Challenge 2009 competition to generate datasets and perform evaluation. In this chapter, we discuss the details of the platform and generated data and our methods to use this data in our implementation. The dataset generation platform contains a challenge client, a dataset generator and a solution checker. The client can invoke the user-implemented composition algorithm as a web service and evaluate its composition time. The solution checker can be used to check the correctness of a given composition solution. The data generator generates web service composition problem in WSDL documents as well as ontology concepts in OWL documents and a set of Web services interfaces in which web service parameters are associated with semantic concepts in OWL files [14]. Here are some important terms which used in this chapter.

- **Concept** is defined in OWL (Web Ontology Language) and it refers to a group of things that share common characteristics.
- **Thing** is defined in OWL (Web Ontology Language). Things are instances of concepts. In addition all things belong to a concept have the same set of attributes
- **Parameters** are part of WSDL (Web Service Definition Language)

- **Service** represent web services defined in WSDL documents. In the datasets generated from WSC (Web Services Challenge) 2009, each service has exactly one port type and each port type has one input message and one output message

To generate a dataset, the user needs to specify some specifications such as the number of services the dataset will have and the number of concepts. Given those parameters, the generator randomly generates a set of given concepts and goal concepts. Then according to those generated concepts as well as the given parameters, it generates a number of paths to form the solutions. Each step of a generated solution contains a set of necessary inputs and a set of desired outputs as well as a set of web services, each of which can independently provide those inputs/outputs. Then, based on the solutions, the generator generates the complete ontology and web service interface set by padding new concepts and services which are not used in the solutions [14]. The generated dataset contains the following files:

- **Services.wsdl** is a file including the description of all generated web services in WSDL.
- **Taxonomy.owl** is a file including all semantic concepts and things that are associated to input and output parameters of web services.
- **Challenge.wsdl** describes the web service composition request. The input parameters of this service are given as known parameters. The output parameters of the service are desired parameters that our algorithm should give.
- **Solution.bpel** includes all possible solutions that exists in the generated dataset.

In our evaluation to use test generator 2009 we applied several techniques for expediting the composition processes. First of all, we parse the given WSDL file and OWL file into our model objects Chapter 6. Then, for each service the subsumtion hierarchy is flattened. Using this technique, we do not need to consider semantic

subsumption during the planning processes. To do that, we use a hash table to index all concepts (defined in an OWL document) that the service takes as inputs or outputs. To generate the composite planning search graph, we need to get a list of currently invokable web services as candidates based on currently known parameters at service composition request. However, the semantic relationship between their I/O parameters need to be known before the composition process. Otherwise, we have to check the relationship map in OWL every time, which is extremely time consuming. As a result, for each of its output parameters, we calculate its directly associated concepts as well as all concepts that subsumes the concept. In addition, for each of its input parameters, we only calculate its directly associated concepts.