REPAIRING WEB SERVICE COMPOSITIONS BASED ON PLANNING GRAPH

Ludeng Zhao

A thesis
in
The Department
of
Computer Science & Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science (Software Engineering)
Concordia University
Montréal, Québec, Canada

April 2010

Your file Votre référence
ISBN: 978-0-494-67211-2
Our file Notre référence
ISBN: 978-0-494-67211-2

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Ludeng Zhao**

Entitled: **Repairing Web Service Compositions Based on Planning Graph**

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

Master of Applied Science (Software Engineering)

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

Signed by the final examining commitee:

_____ Chair

_____ Examiner

_____ Examiner

_____ Examiner

_____ Supervisor

_____ Co-supervisor

Approved _____
Chair of Department or Graduate Program Director

_____ 20 _____ _____

Robin Drew, Dean
Faculty of Engineering and Computer Science

Abstract

Repairing Web Service Compositions Based on Planning Graph

Ludeng Zhao

With the increasing acceptance of service-oriented computing, a growing area of study is the way to reuse the loosely coupled Web services, distributed throughout the Internet, to fulfill business goals in an automated fashion. When the goals cannot be satisfied by a single Web service, a chain of Web services can work together as a "composition" to satisfy the needs. The problem of finding composition plans to satisfy given requests is referred to as the Web service composition problem. In recent years, many studies have been done in this area, and various approaches have been proposed. However, most existing proposals endorse a static viewpoint over Web service composition; while in the real world, change is the rule rather than an exception. Web services may appear and disappear at any time in a non-predictable way. Therefore, valid composition plans may suddenly become invalid due to the environment changes in the business world. In this thesis, techniques to support reparation for an existing plan as a reaction to environment changes are proposed. Approaches of repair are compared to ones of re-planning, with particular attention to the time and quality of both approaches. It will be argued that the approach advocated in this thesis is a viable solution to improve the adaptation of automated Web service composition processes in the context of the real world.

Acknowledgments

I would like to express my deepest gratitude to my supervisor Dr. Yuhong Yan for all her help, support and supervision during the production of this thesis. Without her brilliant ideas and thoughtful advises this thesis would not have been possible. I appreciate Professor Pascal Poizat from University of Evry Val d'Essonne, France for his discussion and support of this work. I also express many thanks to Jamie Johnson for his help during the process of proofreading this thesis. Finally, I would like to thank all my family and friends for their selfless help and support.

I claim that this thesis is from the collaborative research with Dr. Yuhong Yan and Dr. Pascal Poizat. The formalization part in this thesis is from Dr. Yuhong Yan and Dr. Pascal Poizat's work; the algorithm implementation and experiment results in this thesis are developed by me.

Contents

List of Figures viii

List of Tables xi

1 Introduction 1

1.1 Problem and Motivation of Repairing a Web Service Composition . . 1

1.2 Organization of the Thesis 3

2 Background 4

2.1 Web Services 4

2.1.1 Simple Object Access Protocol (SOAP) 5

2.1.2 Web Service Definition Language (WSDL) 6

2.1.3 Business Process Execution Language (BPEL) 7

2.1.4 Web Service Ontology (OWL) and Semantic Extension for WSDL 8

2.2 Formalization for Service Models and Web Service Composition . . . 10

2.2.1 Semantic Models 10

2.2.2 The Problem of Service Composition 13

2.2.3 Web Service Composition as an AI Planning Problem 16

2.3 Automatic Composition Algorithm Evaluation Criteria 17

2.4 Existing Web Service Composition Algorithms 19

2.4.1 BFStar 19

2.4.2 Composition Algorithm Based on Planning Graph 21

3 Planning Graph Repair for Adapting Changes 27

3.1 Introduction 27

3.2 (Re)Planning Algorithm Based on the Planning Graph 28

3.2.1 Main Idea . . . 29
3.2.2 Backward Search for Extracting Plan . . . 29
3.3 Repairing versus Re-planning by Example . . . 31
3.4 The Repairing Algorithms Based on Heuristic Search . . . 31
3.4.1 Motivations . . . 31
3.4.2 Main Idea . . . 33
3.4.3 Heuristic Function . . . 37
3.4.4 Indexing for Repairing Algorithm . . . 41

4 Implementation **44**
4.1 Implement the Web Service Composition Test Bed . . . 44
4.2 Concepts, Things, Services and Parameters . . . 44
4.3 Experiment Setup . . . 45
4.3.1 Dataset Generation . . . 45
4.3.2 Algorithms Invocation Procedures . . . 47
4.4 Building Models . . . 49
4.5 Implementation Details . . . 49
4.5.1 Indexing the Data . . . 49
4.5.2 Flatten the Semantic Relationships . . . 50
4.6 Planning Graph Construction . . . 52
4.7 Backward Search Implementation . . . 53
4.8 Repairing Algorithm Implementation . . . 55
4.8.1 Plan Validator . . . 56
4.8.2 Removing Services from Existing Plans . . . 56
4.8.3 Heuristic Evaluator . . . 57
4.8.4 Repairing Based on Heuristic Search . . . 57

5 Experiments **60**
5.1 Datasets Used in Experiments . . . 60
5.2 Planning Quality Experiments . . . 61
5.2.1 Experiment Method . . . 61
5.2.2 Composition Result . . . 61
5.3 Repairing versus Re-planning Experiments . . . 63
5.3.1 Experiment Method . . . 63

5.3.2 Experiment 1: Remove Web Services from Service Repository 63
5.3.3 Experiment 2: Remove Web Services from Existing Plan . . . 65
5.4 Indexed Repairing versus Re-planning Experiments 67
5.4.1 Experiment Method 67
5.4.2 Fixing Solution 1 WSC Dataset 68
5.4.3 Fixing Solution 2 WSC Dataset 71

6 Conclusion **74**

A Online Source Code Repository **80**

B Planning Graph Algorithm **81**

C Backward Search Algorithm for Extracting Planning Graph **84**

D Heuristic Evaluator **90**

E Repairing Algorithm **92**

F Sample Execution Log of Composition Process using Planning Graph Algorithms **99**

List of Figures

1 Web Service Architecture[Wik10d] 5
2 Layered Structure of SOAP Message[Wik10c] 6
3 Layered Structure of WSDL[Wik10e] 7
4 A Flight Ticket Purchase Service Using BPEL 9
5 Semantic Extension for WSDL[Ble10] 10
6 An Example of OWL in RDF Graph[OWL10] 11
7 Logic Syntax for Pizza OWL Example[OWL10] 12
8 A Web Service Composition Example: Finding Restaurant 13
9 The Composite Web Service for the "Finding Restaurant" Example . 14
10 BFStar Example Lattice[OOLL05] 19
11 Travel example - Planning graph[YPZ10a] 23
12 Planning Graph.(Left): original; (Right): after removal of C2E and D2E 32
13 Planning Graph. (Left): by re-planning; (Right): grown by our repair algorithm 32
14 Insert a new level in partial Planning Graph using "Create new levels at the bottom approach"(a) original (b) Insert A to fix BP_m, (c) Insert A' to fix BP_{m-1} 35
15 Insert a new level in partial Planning Graph using "Create new levels on the top approach"(a) original (b) A can be satisfied by P_{m-1}, (c) add a new level for A 36
16 Example of Heuristic Functions 40
17 Example of Indexing Approach 42
18 Challenge Client (Left) and Data Set Generator GUI (Middle) and Date Set Files Generated (Right) 46
19 The procedure of the Web Service Challenge [Ble10] 48
20 Class Diagram for Model Objects 50

21 Semantic Relationship Between Web Service I/O parameters 52
22 An Example of the Smallest Set 53
23 Repair time with dataset 1 (left) and dataset 2 (middle) and dataset 3 (right) (repair - thick line, re-planning - thin line) 64
24 Number of services in the solutions with dataset 1 (left) and dataset 2 (middle) and dataset 3 (right) (repair - thick line, re-planning - thin line) 64
25 Number of levels in the solutions with dataset 1 (left) and dataset 2 (middle) and dataset 3 (right) (repair - thick line, re-planning - thin line) 65
26 Plan distance to the original solution with dataset 1 (left) and dataset 2 (middle) and dataset 3 (right) (repair - thick line, re-planning - thin line) 65
27 Repair time with dataset 1 (left) and dataset 2 (middle) and dataset 3 (right) (repair - thick line, re-planning - thin line) 65
28 Number of services in the solutions with dataset 1 (left) and dataset 2 (middle) and dataset 3 (right) (repair - thick line, re-planning - thin line) 66
29 Number of levels the solutions with dataset 1 (left) and dataset 2 (middle) and dataset 3 (right) (repair - thick line, re-planning - thin line) . 66
30 Plan distance to the original plan with dataset 1 (left) and dataset 2 (middle) and dataset 3 (right) (repair - thick line, re-planning - thin line) 66
31 Repair time for fixing solution 1 "non-replaceable situation" (left), "replaceable situation" (right) 69
32 Numbers of services in solutions for fixing solution 1 "non-replaceable situation" (left), "replaceable situation" (right) 70
33 Number of levels in solutions for fixing solution 1 "non-replaceable situation" (left), "replaceable situation" (right) 70
34 Plan distance to the original solution (solution 1) "non-replaceable situation" (left), "replaceable situation" (right) 71
35 Repair time for fixing solution 2 "non-replaceable situation" (left), "replaceable situation" (right) 72

36 Numbers of services of solutions for fixing solution 2 "non-replaceable situation" (left), "replaceable situation" (right) 72
37 Number of levels for fixing solution 2 "non-replaceable situation" (left), "replaceable situation" (right) 73
38 Plan distance to the original solution (solution 2) "non-replaceable situation" (left), "replaceable situation" (right) 73

List of Tables

1 Index Table of Replacement Web Services 42
2 Example of Reverse Index Table 51
3 The Originated Web Services for Subgoals 53
4 Data Sets Used in Experiments 61
5 Planning Experiment Results 62

Chapter 1

Introduction

1.1 Problem and Motivation of Repairing a Web Service Composition

Web services are the techniques that are being widely used in today's IT industry. They support machine-to-machine communication, irrelevant to platforms, by exchanging data (in XML format) via the industry standard HTTP protocol [W3C10a] [W3C10f]. Traditional applications can be easily wrapped as Web services to communicate with each other [CFFT06]. Also, the SOA (Service Oriented Architecture) [Wik10b] makes Web services more useful by using them as its basic "building blocks" to form highly scalable and loosely coupled solutions for enterprise applications. In addition, Web services are the ideal technologies for building Semantic Web [W3C10e] due to the nature of their interoperability. However, sometimes a single Web service cannot fulfill given requests completely, and therefore, it needs to cooperate with other Web services to accomplish the requests together. The technique of composing a chain of services to satisfy a given request is commonly known as the Web service composition problem or WSC Problem (Refer to Section 2.2.2). A large amount of research has been done in this area during the past few years, and many algorithms have been proposed. For instance, one of the algorithms we studied is named BFStar[OOLL05]. It applies the well-known A* algorithm from the AI domain to the Web services composition problem. However, most composition algorithms endorse a static viewpoint over Web service composition; while in the real world, change is the rule rather than an exception [YPZ10a]. Web services may appear and disappear

at any time in a non-predictable way. For example, due to user mobility or network failure, the available services change, causing previously valid composition become invalid. Also, users' needs (goals) may also change opportunistically, e.g., when at an airport, a traveler realizes that duty-free shopping is available.

In this thesis, techniques supporting adaptation of a plan as a reaction to business environment changes (e.g. available services or composition requirement changes) are proposed. In essence, this may be achieved in two distinct ways. The first one is to perform a comprehensive *re-planning* from the current execution state of a plan. Re-planning is effectively the same process as planning, but with the addition of taking the updated parameters into account. Another way is trying to *repair* the plan in response to changes, reusing most of the existing plan whenever possible. We believe plan repair is a valuable solution to the WSC Problem in a changing world because:

1. It makes it possible to retain the effects of a partially executed plan, which is far more time and cost efficient than throwing everything away and rolling back its effects;

2. Even if, in theory, modifying an existing plan is no more efficient than a comprehensive re-planning in the worst case [vdKdW05], it is expected that plan repair is, in practice, often more efficient than re-planning, since a large part of an original plan is usually still valid after a change has occurred;

3. Commitment to the unexecuted services can be kept as much as possible, which can be mandatory for business/security reasons;

4. Finally, the end-users typically prefer to use a repaired plan that resembles the original rather than a very different plan for business/security reasons.

Our study is based on planning graphs [KPL97]. Planning graphs enable a compact representation of relations between Web services, and model the whole problem world. Even with some changes, part of a planning graph is still valid. Therefore, we believe that the use of planning graphs can be more beneficial than other techniques in solving problems of plan adaptation. In our approach, we first identify the new composition problem with the new set of available Web services and new goals. The disappeared (or damaged) Web services are removed from the original planning graph and new goals are added to the goals level. This yields a partial planning

graph. The repairing algorithm "regrows" this partial planning graph wherever the heuristic function tells that the unimplemented goals and broken preconditions can be satisfied. Its objective is not to build a full planning graph, but to quickly search for a feasible solution, while maximally reusing whatever is in the partial graph. Compared to re-planning, the repair algorithm constructs only a portion of a full planning graph until a solution is found. It can be faster than re-planning, especially when the composition problem does not change too much and a large part of the original planning graph is still valid. In our experiments, we have demonstrated this. Our experiments also show that the solutions from repairs have the similar quality as those from re-planning. However, in some cases, our repairing algorithm may not find existing solutions while re-planning would, which is the trade off of the speed.

As far as Web services are concerned, we take into consideration their WSDL interfaces extended with semantic information for inputs and outputs of operations. We suppose Web services are stateless, which means a Web service does not distinguish users nor maintain states between requests. A Web service can be considered as a function that accepts a set of data and output a set of data, therefore we do not consider the internal behaviour of Web services. Accordingly, composition requirements are data-oriented and not based on some required conversation. These choices are consistent with many approaches for WSC Problem; e.g. [OLK07], [OOLL05] and [HM07]. More importantly, it suits the Web Service Challenge [Ble10] which enables us to evaluate our proposal on large-scale data sets.

1.2 Organization of the Thesis

The remaining of this thesis is structured as follows: in Chapter 2 we briefly introduce necessary background information as well as previous works that have been done in this domain. Later in the chapter, we present our formalization approach for Web service composition problems. Based on the formal models, we propose our algorithms for both re-planning and repairing Web services in Chapter 3. Chapter 4 covers the details related to algorithm implementation, and Chapter 5 shows the experiment results of our works. Finally, we conclude our works in Chapter 6.

Chapter 2

Background

2.1 Web Services

Given the definition by the World Wide Web Consortium (W3C) [W3C10g], "a Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with Web services in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards". In today's real world, Web services are frequently just a set of application programming interfaces (API) or web APIs that can be accessed over networks, such as the Internet [Wik10d]. Depending on the specification of how it is applied, a Web service can be categorized into either "Traditional Web Service" as it is defined by W3C or "Representational State Transfer (RESTful) Web Service" [Wik10a] which uses the standard HTTP methods including "PUT", "GET" and "DELETE" that are supported by today's Internet. In this thesis, we focus on the Web services standard that is defined by W3C. The term "Web service" discussed in this thesis means Web services that follow the WSDL and SOAP specifications.

Figure 1 shows the basic architecture of Web services. Web services, depending on their roles, can be referred to as either "service requesters" or "service providers".

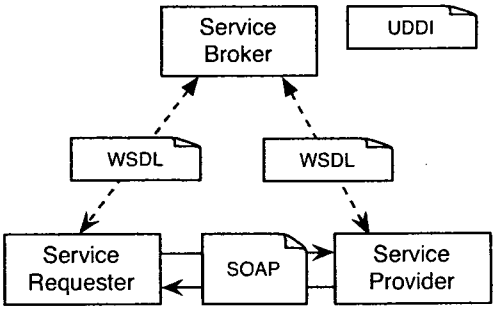

Figure 1: Web Service Architecture[Wik10d]

Each "service provider" is associated with a WSDL document which contains information related to the services it contains and the way to invoke those services. "Service provider" publishes its own WSDL to a "service broker" using UDDI specification [OAS10b] to allow "service requester" to locate the services that they want to use. After the "service requester" locates the service, it communicates with the "service provider" via SOAP protocol with the address that is specified in WSDL document. The details of related specification are discussed in following sections.

2.1.1 Simple Object Access Protocol (SOAP)

SOAP [W3C10f] is a protocol specification for exchanging structured information (such as XML) between Web services. It is usually built upon HTTP which is a standard protocol in today's Internet infrastructure. Therefore, a SOAP message can be carried as the payload of an HTTP message. Due to the nature of using these widely accepted technologies, there are some obvious benefits to use SOAP to exchange data:

Platform independent and language independent: instead of encoding information into platform relevant binary format, SOAP uses XML as its information carrier. Almost every platform and programming language is able to operate XML files without interpreting. However, one thing worth noticing is that SOAP has more overhead compared to binary format, which might lower the efficiency in some cases.

Firewall transparency: most firewalls allow the transmission via HTTP ports. To the firewall, SOAP is a branch of XML data traveling through HTTP and thus is "transparent".

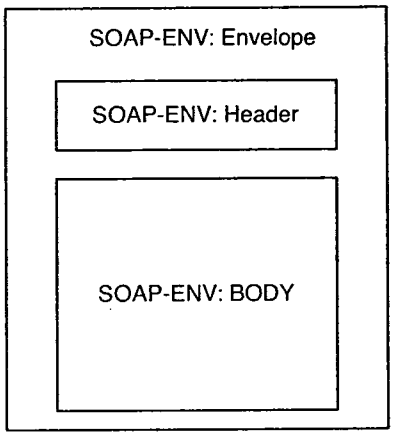

Figure 2: Layered Structure of SOAP Message[Wik10c]

As Figure 2 shows, a SOAP message contains three components: "Envelope", "Header" and "Body". An "Envelope" is just a pair of XML tags that mark the beginning and ending of a SOAP message. The "Header" contains the important information for delivering the message, such as routing policy, security policy and QoS policy. The "Header" is sometimes optional. The "Body" is where the real data resides.

2.1.2 Web Service Definition Language (WSDL)

Web Service Definition Language or WDSL [W3C10h] is an XML based language providing a mechanism to describe Web services. Every Web service is associated with one WSDL document so that other Web services or applications can use it to understand what services that Web service provides and the way to invoke those services. WSDL defines Web service as a collection of "ports" (WSDL 1.1) or "end-points" (WSDL 2.0) (see Figure 3). Each "port" is associated with a network address with reusable bindings. And each "port" has a corresponding "port type" (WSDL

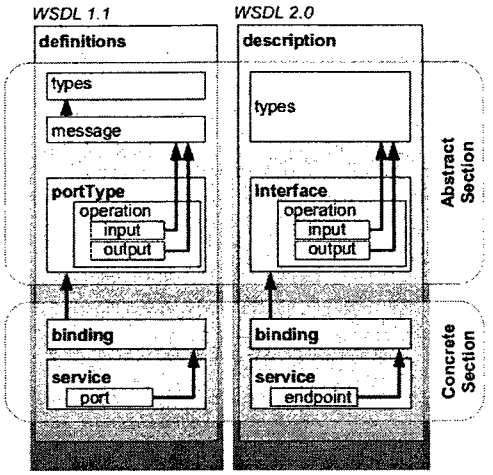

Figure 3: Layered Structure of WSDL[Wik10e]

1.1) or "interface" (WSDL 2.0), in which it defines supported operations of one Web service. Each "operation" can have one "input message" and one "output message". The format of these messages is defined at "message" sections and the data types used are defined in the "types" sections.

2.1.3 Business Process Execution Language (BPEL)

Business Process Execution Language or BPEL [OAS10a] is a language for describing business process activities as Web services, defining how they can be composed to accomplish specific tasks. A business process is a set of coordinated tasks and activities that will lead to the accomplishment of a specific organization goal. In BPEL, tasks and activities can be represented as Web services. Web services provide a basic stateless transaction model based on message exchange. However, Web services themselves are not suitable to describe today's complex business interactions since they have no knowledge about who they are interacting with. To address this issue, BPEL extends the Web service specification by providing a mechanism to support stateful and long-running business processes. The participated Web services in a BPEL process are named "Partners" and have "Partner Links" which describe the roles of participated Web services in business processes. Alongside this, BPEL also

introduces many programming language-like syntax such as "ifelse", "while", "for" and "switch" to manage the logic flows of business processes. And, it allows users to create, copy and use variables for storing the information related to business processes and thus can support complex processes. Figure 4 shows a simple BPEL process representing a flight ticket purchase service. In order to purchase a ticket, the user needs to send his/her "username", "password" and "desired flight" to the purchase service. The purchase service then interacts with two external Web services named: "Login Service" and "Search Flight Service" to validate the users' login information as well as the availability of the desired flights. If both services return "true" then the purchase service records the purchase and returns "purchase successful" to users. Otherwise, it will return "false".

2.1.4 Web Service Ontology (OWL) and Semantic Extension for WSDL

By default, a Web service defined in WSDL does not support the semantic invocation processes. For example, a Web service defined in WSDL does not know the two parameters "First Name" and "Given Name" have the same meaning since it only checks the syntax difference of the given parameters. Also, the Web service has no such knowledge to tell that both "ISBN-8" and "ISBN-16" are "ISBN". In order to facilitate the automatic Web service composition process in the real world context, we have to extend the current WSDL specification by adding semantic extensions as is suggested in [Ble10]. By doing this, we can associate the syntax parameter defined in a WSDL to its semantic meaning defined in an OWL (Web Ontology Language [W3C10b]) file. Therefore, a Web service can check the semantic meaning of its parameters as well as their relationship by checking the OWL file. As Figure 5 shows, a section named "semExtension" is added to the end of WSDL file, in which it points the parameter named "price" to the semantic definition "Bookprice" located in an OWL file.

Figure 6 shows an OWL example represented in an RDF (Resource Description Framework [W3C10c]) graph. It represents the logics shown in Figure 7. We propose an approach in Section 2.2 for converting semantic problems into syntactic problems in order to facilitate our algorithm.

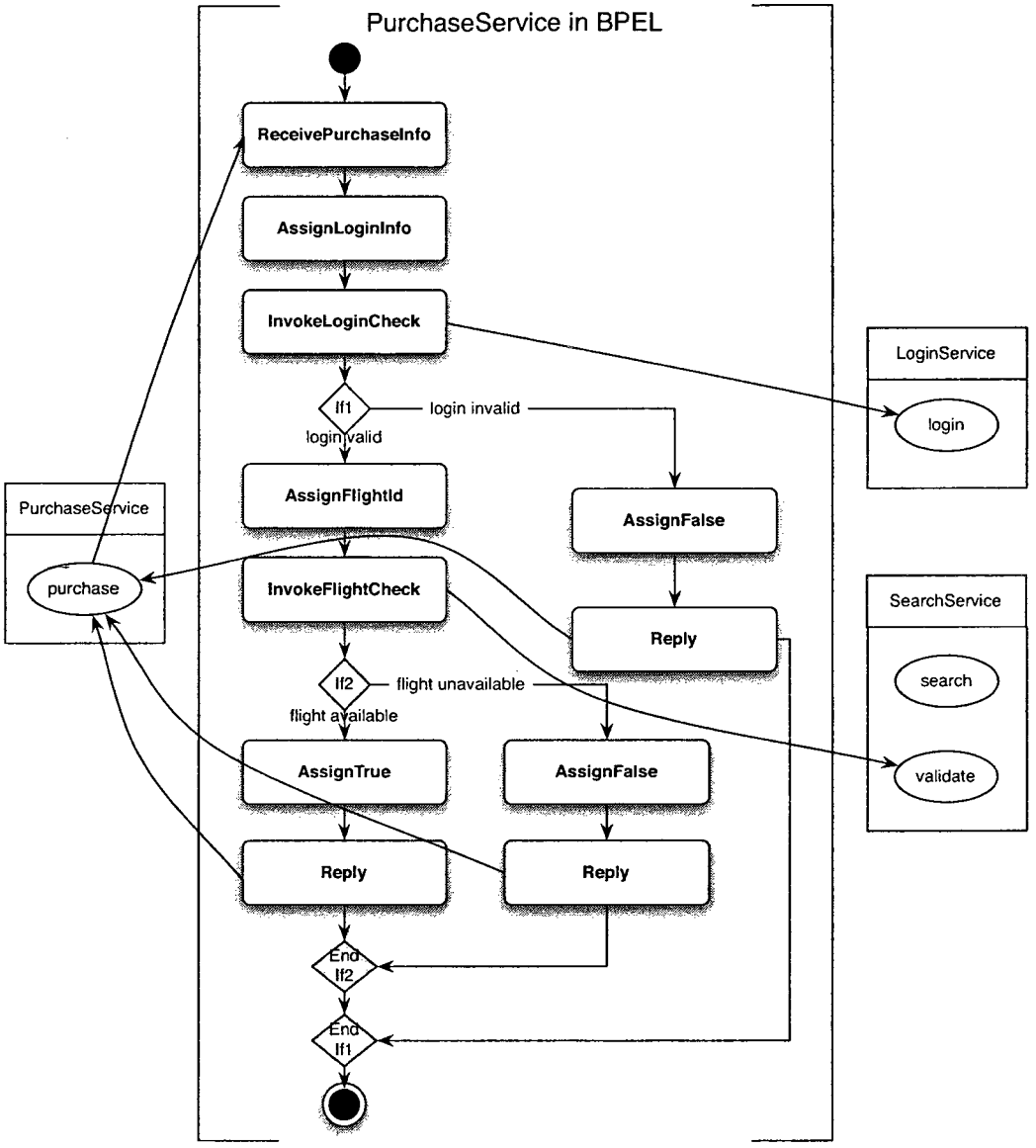

Figure 4: A Flight Ticket Purchase Service Using BPEL

```
<mece:semExtension>
    <!-- Semantic Extension for the message with ID "getPriceRequest" -->
    <mece:semMessageExt id="BookShopARequestMessage">
        <!-- Smantic Annotation for the xsd:element with ID "price" -->
        <mece:semExt id="price">
            <!-- Ontology reference for the semantic individual for this element -->
            <mece:ontologyRef>
                http://www.owl-ontologies.com/Ontology.owl#Bookprice
            </mece:ontologyRef>
        </mece:semExt>
    </mece:semMessageExt>
    <!-- Arbitrary amount of message annotations -->
    <mece:semMessageExt id="BookShopAResponseMessage"/>
    ...
    <mece:semMessageExt .../>
    ...
</mece:semExtension>
```

Figure 5: Semantic Extension for WSDL[Ble10]

2.2 Formalization for Service Models and Web Service Composition

In this Section we present first our formal models for services and user needs (or requirements) as we discussed in other papers: [YPZ10a] and [YPZ10b]. Different approaches have been followed to address Web service composition. We adopted an AI planning point of view, which has been demonstrated as being very efficient for automatic Web service composition. This will have an impact on our formal models, and accordingly, on the composition algorithm we present in the sequel of this thesis.

2.2.1 Semantic Models

In order to enable automatic service discovery and composition from user needs, some forms of semantics have to be associated to services. Basically, the names of the services' provided operations could be used (i.e. operation name such as "$findHotelByName$"), but one can hardly imagine and, further, achieve interoperability at this level in a context where services are designed by different third parties.

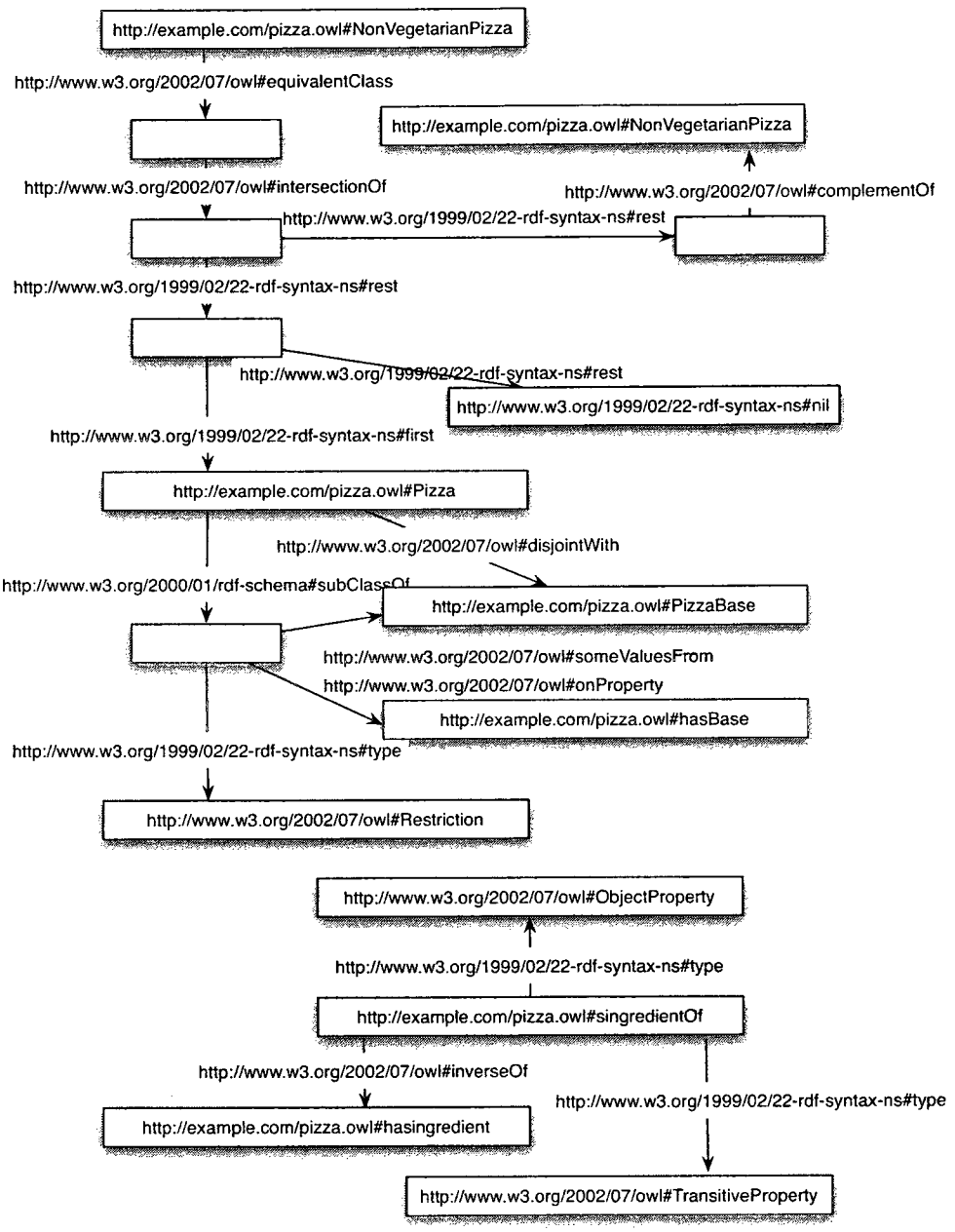

Figure 6: An Example of OWL in RDF Graph[OWL10]

$$\text{PIZZA} \sqsubseteq \exists \text{ hasBase.PIZZABASE}$$
$$\text{PIZZA} \sqcap \text{PIZZA BASE} \equiv \bot$$
$$\text{NONVEGETARIANPIZZA} \equiv \text{PIZZA} \sqcap \neg \text{VEGETARIAPIZZA}$$
$$Tr(\text{isIngredientOf})$$
$$\text{isIngredientOf} \equiv hasIngredient^{-}$$

Figure 7: Logic Syntax for Pizza OWL Example[OWL10]

Adaptation approaches [HBM08] [SEG08], have proposed to rely on so-called adaptation contracts that must be given (totally or partly) manually. To avoid putting this burden on the user, semantics based on shared ontologies may be used instead to provide fully-automatic compositions. Services may indeed convey two kinds of semantic information. The first one is related to data that is transmitted along with messages. For example, a Web service providing hotel registration can be semantically described as receiving semantic information "*hotelname*", "*username*", "*arrivaldate*", "*leftdate*" as the inputs and providing a "*hotelregistration*" as the output. The second way to associate semantic information to services is related to functionalities, or capacities, that are fulfilled by services. Provided each service has a single capacity, this can be treated as a specific output in our algorithms, e.g., the above mentioned service could be described with an additional semantic output, "*hotelbookingcapacity*", or we can even suppose its capacity is self-contained in its outputs (here, "*hotelregistration*").

In our approach, semantic information is supported with a structure called *Data Semantic Structure* [BP08] (Definition 1)

Definition 1 *A Data Semantic Structure (DSS) is a couple* (D, R^D) *where* D *is a set of concepts that represent the semantics of some data, and* $R^D = \{R_i^D : 2^D \leftarrow 2^D\}$ *is a set of relations between concepts.*

The members of R^D define how, given some data, other data can be produced. Given two sets, $D_1, D_2 \subseteq D$, we say that D_2 can be obtained from D_1, and we write $D_1 \rightarrow_{R^D} D_2$ or simply $D_1 \rightarrow D_2$, when $\exists R_i^D \in R^D, R_i^D(D_1) = D_2$. Some instances of the R_i^D relations - representing composition, decomposition and casting - together with implementation rules are presented in [BP08]. Here we propose a generalization of these as presented in [YPZ10a].

The interface (operations and types of exchanged messages) of a Web service is described in a WSDL [W3C10h] file. DSSs are a subclass of what can be described using OWL [W3C10b], a very expressive language for describing super(sub)-classes, function domains and rages, equivalence and transition relations, etc. We can extend OWL for other relations as well, e.g., a parameter can correspond to the (de)composition of a set of parameters according to some functions. WSDL can be extended to reference OWL semantic annotations [Ble10]. The SAWSDL standard [W3C10d] can also be used for this.

2.2.2 The Problem of Service Composition

An Example of Web Service Composition Problems and Solutions

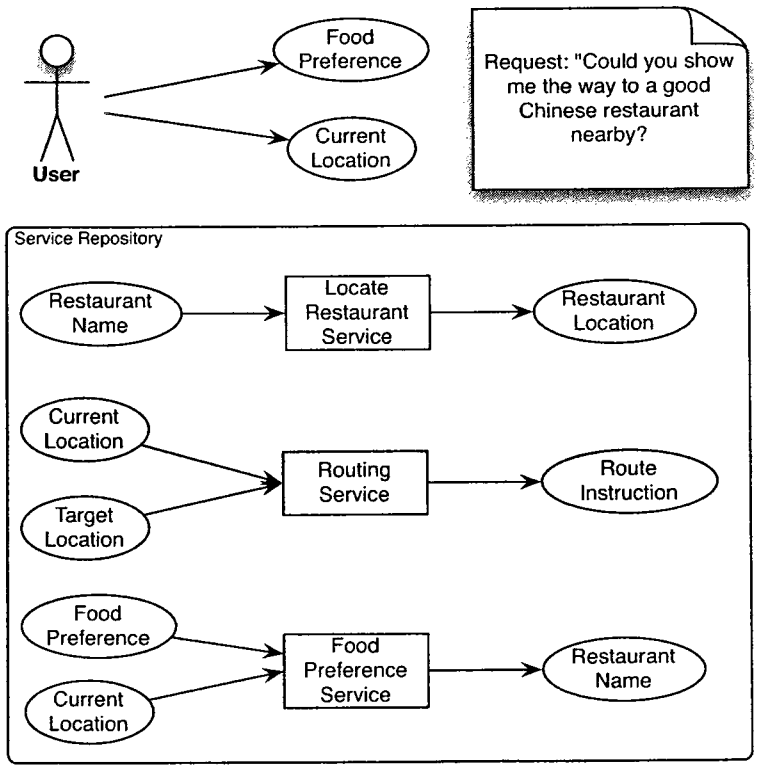

Figure 8: A Web Service Composition Example: Finding Restaurant

Consider such a scenario (see Figure 8): A Web service developer would like to develop a Web service which can guide users to the best restaurant nearby that has his/her favourite food style. An example of such query might be "Could you show me the way to a good Chinese restaurant nearby?" So the developer looks up a service repository which is publicly available, he finds that no single service in the repository can fully satisfy such a request. However, there exist three Web services which can be used together as a whole to satisfy the request. So he decides to build a composite Web service that uses those three Web services (shown in Figure 9).

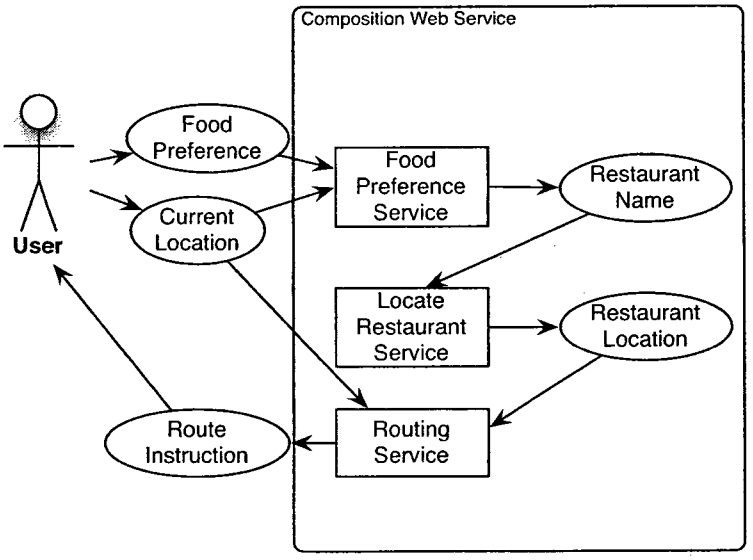

Figure 9: The Composite Web Service for the "Finding Restaurant" Example

Although the Web service composition solution can be designed manually by developers, in this paper we mainly focus on an automated way for designing composition Web services. When users submit their queries to a "Web service composition engine" the automated algorithm should find a composition plan and return the request information to users. This approach gives the possibility of doing Web service compositions on large-scale service repositories (e.g. the Internet). Additionally, we also make following assumptions about Web services:

1. We suppose that Web services are stateless and do not feature a behavioural

description specifying in which ordering operations are to be called. Hence, without loss of generality, we have supposed each Web service has only one operation. Whenever a Web service has more than one, we can use indexing, e.g., service w with operations o_1 and o_2 becomes services $w.o_1$ and $w.o_2$;

2. Some Web service input parameters are *informatics* (i.e. non-consumable), and can be therefore used without limitation. However, other Web service input parameters are consumable, meaning that they can be used only once. For example, given an order processing service, orders are consumable: once processed, orders cannot be used by other services. OWL's rdf:property can be extended to describe parameters' consumability properties.

Model of the Service Composition Problem

Definition 2 *Being given a DSS(D, R^D), a Web service w is a tuple (in, out) where $in \subseteq D$ denotes the input parameters of w and $out \subseteq D$ denotes the output parameters of w. We denote $in^- \subseteq in$ the set of input parameters that are consumed by the service.*

A Web service can be considered for composition as a function that takes input(s) and return output(s) (Definition 2). We abstract these input(s) and output(s) associated to the Web service operations as semantic data, called parameters. An example of this is where relations are defined between WSDL message types and service model parameters through the use of XPath can be found in [MPM08].

DSS supports data transformation using relations in R^D. This enables service compositions where mismatch would usually prevent it. For the sake of uniformity, being given a DSS(D, R^D), for each R_i^D in R^D, for each $D_1 \rightsquigarrow_{R_i^D} D_2$, we define a data adaptation service $w_{R_i} = (D_1, D_2)$. Such a service can be automatically implemented, either as a service or as a reusable standalone piece of code in any implementation language supporting XPath and assignment such as WS-BPEL.

If the output parameters of a set of services can produce at least one of the input parameters of another service, we say they can be connected (Definition 3).

Definition 3 *Assume every parameter in the parameter set $D_W = \{d_1, d_2, ..., d_k\}$ is an output parameter of one of the Web services in a set $W = \{w_1, w_2, ..., w_m\}$,*

i.e., $W = \{w_i | \exists d_j \in D_W, d_j \in out(w_i), i = 1, ..., m\}$. *If* $\{d_1, d_2, ..., d_k\} \rightsquigarrow \{d_n\}$, *and* $d_n \in in(w_n)$, *then every service in* W *can be connected to* w_n, *annotated as* $w_i \rhd w_n$.

Finally, the last input for any WSC algorithm is the description of the composition requirements corresponding to the user needs (Definition 4).

Definition 4 *Being given a* $DSS(D, R^D)$, *a composition requirement is a couple* (D_U^{in}, D_U^{out}) *where* $D_U^{in} \subseteq D$ *is the set of provided (or input) parameters and* $D_U^{out} \subseteq D$ *is the set of required (or output) parameters.*

The objective of a WSC algorithm may now be formally described. Given a set of available Web services, a structure (DSS) describing the semantic information that is associated to the services, and a composition requirement, service composition is to generate a connected subset of the services that satisfies the composition requirement (Definition 5).

Definition 5 *A composition requirement* (D_U^{in}, D_U^{out}) *is satisfied by a set of connected Web services* $W = \{w_1, ..., w_n\}$ *iff,* $\forall i \in \{1, ..., n\}$:

- $\forall d \in in(w_i), D_U^{in} \cup out(w_1) \cup ... \cup out(w_{i-1}) \rightsquigarrow d$ *and*
- $\forall d \in D_U^{out}, D_U^{in} \cup out(w_1) \cup ... \cup out(w_n) \rightsquigarrow d$

A composition requirement is satisfied iff there is at least one set of connected services satisfying it.

2.2.3 Web Service Composition as an AI Planning Problem

AI planning has been successfully applied to solve the WSC problem through its encoding as a planning problem [Pee05] [MP09]. The following definitions in this section are modified from [GNT04].

Definition 6 *A planning problem is a triple* $P = ((S, A, \gamma), s_0, g)$, *where*

- S *is a set of states, with a state* s *being a subset of a finite set of proposition symbols* L, *where* $L = \{p_1, ..., p_n\}$.

- *A is a set of actions, with an action a being a triple* $(precond, effects^-, effects^+)$ *where precond(a) denotes the preconditions of a, and* $effects^-(a)$ *and* $effects^+(a)$, *with* $effects^-(a) \cap effects^-(a) = \emptyset$, *denote respectively the negative and the positive effects of a.*

- γ *is a state transition function such that, for any state s where* $precond(a) \subseteq s, \gamma(s,a) = (s - effects^-(a)) \cup effects^+(a)$.

- $s_0 \in S$ *is the initial state.*

- $g \subseteq L$ *is a set of propositions called* goal propositions *(or simply goal).*

- *A plan is any sequence of actions* $\pi =< a_1, ..., a_k >$, *where* $k \geq 0$.

The WSC problem can be mapped to a planning problem as follows:

- Each service, w, is mapped to an action with the same name, w. The input parameters of the service, $in(w)$, are mapped to the action's preconditions, $precond(w)$, and its output parameters, $out(w)$, are mapped to the action's positive effects, $effect^+(w)$. Consumable parameters, $in^-(w)$ are also mapped to negative effects, $effects^-(w)$.

- The input parameters of the composition requirement, D_U^{in}, are mapped to the initial state, s_0

- The output parameters of the composition requirement, D_U^{out}, are mapped to the goal, g.

2.3 Automatic Composition Algorithm Evaluation Criteria

For those who aim to design algorithms for a Web service composition system there are several objectives [YPZ10a]:

The composition time

The composition time should be as short as possible. Composition time is the time period from the start of the composition process to the end of the composition process.

It represents the time cost needed for a Web service composition system in response to the users' requests. In practical terms, the composition solution might be void if the composition time is too long, because the environment is likely changed during that period.

The number of Web services

The number of Web services in the composition solution should be as few as possible because more Web services usually mean more costs. In the real world, the cost could be the cost of money, labour and resources.

The number of levels

The number of levels in the composition solution should be as few as possible. Usually few levels mean lower cost in terms of execution time. This is because some services can be executed in parallel without waiting for other Web services to complete.

The redundancy rate

It is possible that a composition solution can have redundant Web services that if these Web services are removed, the composition solution is still valid and satisfies the goals. The redundancy rate is the percentage of redundant Web services in the composition solution and should be keep as low as possible.

The plan distance

Plan distance is defined in Definition 7 as in [FGLS06]:

Definition 7 *Given an original plan, π_0, and a new plan π_1, the difference between π_0 and π_1, $D(\pi_0, \pi_1)$, is the number of actions that appear in π_1 and not in π_0 plus the number of actions that appear in π_0 and not in π_1.*

We are interested in achieving low plan distance because we prefer that the new plan is similar to the original one. In the Web service composition context, this means that we can keep our promise to the business partners in the original plan.

2.4 Existing Web Service Composition Algorithms

2.4.1 BFStar

The BFStar algorithm was first proposed by Seog-Chan Oh in [OOLL05]. It divides the Web service composition problem into two separate sub-problems: (1) A fast and efficient way for doing membership checking (i.e. what Web services can be invoked based on currently known information). The Bloom filter [Blo70] is applied for the membership checking to minimize the cost of finding candidate Web services in each execution step. Although a false positive is possible, false negatives are not. (2) A fast and efficient search algorithm based on AI technology. The BFStar algorithm (Algorithm 1) with different heuristic functions is used for finding the goal parameters. The main idea of the BFStar algorithm is letting the searching algorithm use heuristic functions to evaluate each of its candidates. The one with the highest heuristic score will be added into the solution.

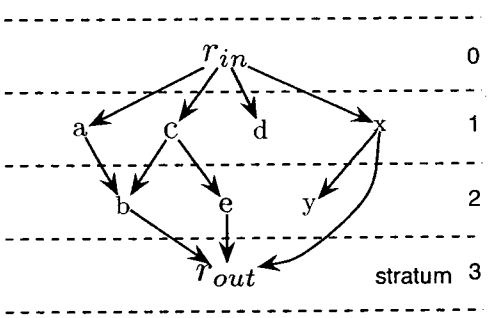

Figure 10: BFStar Example Lattice[OOLL05]

As is shown in Figure 10, starting from r_{in}, one has 4 choices to make in sequential mode. Then, BFStar will suggest only 1 out of 4 as a next web service to visit based on heuristics. Suppose d was visited. Then, from d, again all possible next choices are computed and one of them is suggested. However, in this case, there is no available next choice to make, thus one has to backtrack to the previous state. Then, another one, say x, out of 4 is suggested as a next move, and so on. When next move is the goal state, i.e., r_{out}, the search is successful.

Due to the nature of being forward-searching, the quality of the composition result solely depends on its heuristic function. Several varieties that have slightly different heuristic functions are also proposed along with BFStar Algorithm. For instance, consider one such variety, named "BFStar Lookahead"; when evaluating one candidate, it performs a "lookahead" by evaluating the children candidates of that candidate. The final heuristic score of the candidate is equal to the candidate's own score plus the best score of its children. This approach increases the accuracy of selecting candidates in some cases (e.g. all candidates in current level have the same scores while their children have different scores). However, the trade-off is that the composition time is significantly increased by performing "lookahead".

Algorithm 1 BF* Algorithm

Input: All Web services W

Output: A path: $r_{in} \Rightarrow ... \Rightarrow r_{out}$

1: $\Omega \leftarrow \emptyset$ and $\Sigma \leftarrow r_{in}$;
2: print r_{in},"$\Rightarrow$"
3: **while** $\Sigma \not\supseteq r_{out}$ **do**
4: $\quad \delta \leftarrow \{w | w \in W, W \notin \Omega, w_{in} \subseteq \Sigma\}$
5: $\quad w^{min} \leftarrow w(\in \delta)$ with MIN($f(w)$);
6: $\quad \Omega \leftarrow \Omega \cup w^{min}$
7: $\quad \Sigma \leftarrow \Sigma \cup w^{min}_{out}$
8: $\quad$ print w^{min}, "$\Rightarrow$"
9: print r_{out}

Heuristic functions for standard BF* Algorithm:

$$h(n) = 1/|(r_{out}/\Sigma) \cap n_{out}| \tag{1}$$

$$g(n) = |\Omega| \tag{2}$$

Where the r_{out} is the set of goals from the composition query; Σ is the set of currently known parameters; n_{out} is the outputs of the given Web service n and Ω is the set of Web services added in the plan.

Heuristic functions for lookahead BF* Algorithm:

$$h(n) = 1/|(r_{out}/\Sigma) \cap (n_{out} \cup c(n)_{out})| \tag{3}$$

Where $c(n)_{out}$ is the child of n_{out} with the maximum contribution in that:

$$c(n)_{out} = MAX\{w \in \delta^{'}|(r_{out}/(\Sigma \cup n_{out}) \cap w_{out})\} \tag{4}$$

$$\delta = \{w|w \in W, w \notin \Omega, w_{in} \subseteq (\Sigma \cup n_{out})\} \tag{5}$$

The strength of BFStar is that it is able to get the composition results in a reasonable time due to the benefits from using the heuristic function and the Bloom filter. However, almost as significant as its strength, it inherits the drawbacks from using an A* search algorithm: the composition result is in sequential order instead of parallel. It can only invoke one Web service at each execution step even though some Web services can be invoked in parallel; this lowers its execution efficiency, especially when the number of services need to be invoked is large.

2.4.2 Composition Algorithm Based on Planning Graph

Different planning algorithms have been proposed to solve planning problems, *e.g.*, depending on whether they are building the graph structure underlying a planning problem in a forward (from initial state) or backward (from goal) manner. As was proposed by Xiangrong Zheng and Yuhong Yan in their works [ZY08] and [YZ08], the planning graph can be applied to the WSC Problem. In this thesis, we carried on their works and applied the same formalization methods to model the service composition problem. Therefore, it is important to introduce the planning graph here.

Definition 8 *In a planning graph, a* ***layered plan*** *is a sequence of sets of actions* $< \pi_1, \pi_2, ...\pi_n >$, *in which each* $\pi_i (i = 1, ..., n)$ *is independent (see Definition 9).* π_1 *is applicable to* s_0. π_i *is applicable to* $\gamma(s_{i-2}, \pi_{i-1})$ *when* $i = 2, ..., n$.
$g \subseteq \gamma(...(\gamma(\gamma(s_0, \pi_1), \pi_2)...\pi_n)$

A planning graph iteratively expands itself one level at a time. The process of graph expansion continues until either it reaches a level where the proposition set contains all goal propositions or a fixed point level. The goal cannot be attained if the latter happens first. Definition 9 defines the independence of two actions. Two

actions can also exclude each other due to the conflicts of their effects. We can add both independent and dependant actions in one action layer in the planning graph. However, two exclusive actions cannot appear in the same plan. The planning graph searches backwardly from the last level of the graph for a solution. It is known that a planning graph can be constructed in polynomial time.

Definition 9 *In a planning graph, two actions a and b are* ***independent*** *iff they satisfy* $effect^{-}(a) \cap [precond(b) \cup effect^{+}(b)] = \emptyset$, *and* $effects^{-}(b) \cap [precond(a) \cup effects^{+}(a)] = \emptyset$. *A set of actions is independent when its actions are pairwise independent*

Notice that although we can model negative effects we do not consider the situations of negative effects in the rest of this thesis.

In order to demonstrate the use of planning graph for the Web service composition problem, we give a simple travelling example as it shows below:

An Example of using Planning Graph for WSC

First we introduce the DSS for our example. The concepts we use correspond to:

- users: "uinfo"(made up of "uname" and "ucity"),
- departure and return dates: "fromdate" and "todate",
- departure and return cities (resp. countries): "depcity" and "depcountry" (resp. "destcity" and "destcountry")
- travelling alerts: "travelalert"
- flight request: "flightreq"
- registration information for planes and hotels, "planereg" and "hotelreg"

Additionally, different relations exist between these concepts:

- subsumption: from "ucity" to "depcity" and from "travelcity" to "destcity",
- decomposition: from "uinfo" to "uname" and "ucity"
- composition: from "depcity", "destcity", "fromdate" and "todate" to "flightreq"

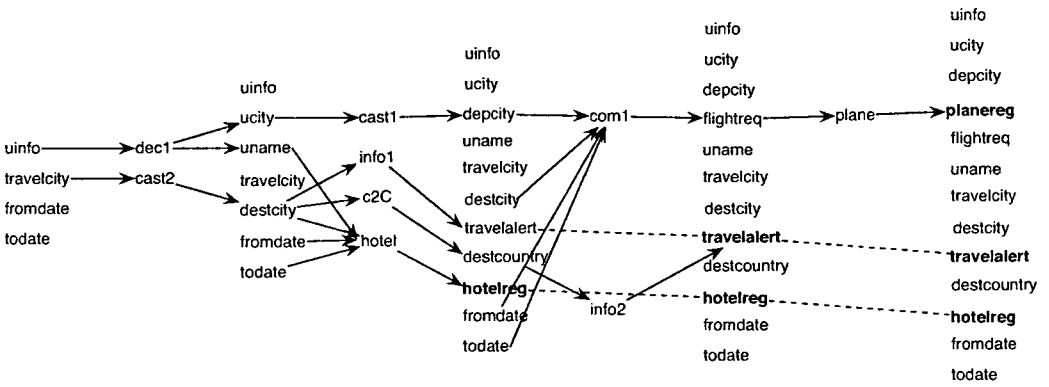

Figure 11: Travel example - Planning graph[YPZ10a]

As explained before, these four relations will be supported by data adaptation services (respectively cast1, cast2, dec1 and comp1) that can automatically be implemented. The repository of available services contains the following services:

- info1({destcity},{travelalert}),
- info2({destcountry},{travelalert}),
- c2C({destcity},{destcountry}),
- plane({flightreq,uname},{planereg}),
- hotel({travelcity, fromdate, todate, uname}, {hotelreg}).

Finally, the user request is: ({uinfo, travelcity, fromdate, todate}, {planereg, hotelreg, travelalert}).

We can build a planning graph as in Figure 11. Identity links (when some data is kept for the next data layer) are not represented for clarity, but for required parameters (dashed lines).

From this planning graph, backtracking from the required data (in bold), we can compute two solutions (plans) as is shown below:

Solution 1: (dec1 | cast2); (cast1 | info1 | hotel); com1; plane

Solution 2: (desc1 | cast2); (cast1 | c2C | hotel); (com1 | info2); plane

We use ";" to represent the sequence of service calls and "|" to represent service calls in parallel (i.e. "flow" in BPEL). One may notice that without the adaptation enabled by DSS relations (cast1, cast2, dec1 and comp1), no composition would be possible at all.

Main Idea of Planning Graph Algorithm

As is introduced in [GNT04], the planning graph algorithm performs a procedure close to *iterative deepening*, discovering a new part of the search space at each iteration. It iteratively expands the planning graph by one level, then it searches backward from the last level of this graph for a solution (Section 3.2). Algorithm 2 shows the forward expanding process expands the planning graph by creating a new action level A_i and a new proposition level P_i. Then, as is shown in Algorithm 3, the expanding process adds all Web services whose preconditions are satisfied to A_i and adds their outputs to P_i. The expanding proceeds to a level P_i in which all of the goal propositions are included or when $P_i = P_{i-1}$, named "fixedpoint" [ZY08]. The works of [ZY08] and [YZ08] extends the expanding process with a set of predefines rules, named "strategies", in order to keep redundant Web services from being included during the process. Backward search is used to extract the solution from the planning graph constructed by the expanding process. The details of backward search are discussed in Section 3.2.

Algorithm 2 *Compose*(A, s_0, g)

Notes about the algorithm: $G =< P_0, A_1, P_1, ..., A_i, P_i >$ is a simplified planning graph

1: **repeat**
2: $G \leftarrow StandardExpand(G)$
3: $i = i + 1$
4: **until** $g \subseteq P_i \vee Fixedpoint(G)$
5: **if** $g \subseteq P_i$ **then**
6: $Output(< A_1, A_2, ..., A_3 >)$
7: **else**
8: $Output(\emptyset)$
9: **if** $Fixedpoint(G)$ **then**
10: print("Reach fixed point")

Algorithm 3 *StandardExpand*$(< P_0, A_1, ..., A_i, P_i >)$

1: $A_i \leftarrow \{a \in A | precond(a) \subseteq P_{i-1}\}$
2: $P_i \leftarrow \{p | \exists a \in A_i : p \in effects(a)\}$
3: **for** each $a \in A_i$ **do**
4: link a with precondition arcs to $precond(a)$ in P_{i-1}
5: link a with to each of its $effects(a)$ in P_i
6: **return** $< P_0, A_1, ..., A_i, P_i >$

Algorithm 4 *Fixedpoint*$(< P_0, A_1, ..., A_i, P_i >)$

1: **if** $P_i = P_{i-1}$ **then**
2: **return true**
3: **else**
4: **return false**

Algorithm 5 *ExpandWithStrategies*$(< P_0, A_1, ..., A_i, P_i >)$

Notes about the algorithm: $valid(a)$: action a can be used in the algorithm; $invalid(a)$: action a can not be used in the algorithm again

1: **for** $a \in A$ **do**
2: **if** $precond(a) \in P_i \wedge valid(a)$ **then**
3: $A_{i+1} \leftarrow A_{i+1} \cup a$
4: $invalid(a)$
5: $P_{i+1} \leftarrow P_i$
6: **for** $a \in A_{i+1}$ **do**
7: **if** $effect(a) \not\subseteq P_{i+1}$ **then**
8: $P_{i+1} \leftarrow P_{i+1} \cup effects(a)$
9: **else**
10: $A_{i+1} \leftarrow A_{i+1}/a$
11: $invalid(a)$
12: **if** $g \subseteq P_{i+1}$ **then**
13: discard all the other actions in A_{i+1} break
14: **for** a in A_i **do**
15: **if** $effects(a) \cap precond(a) \mid a \in A_{i+1} = \emptyset$ **then**
16: $A_i \leftarrow A_i/a$
17: $valid(a)$

Chapter 3

Planning Graph Repair for Adapting Changes

3.1 Introduction

WSC problem should be considered in the open world. An open world is always changing. The changes would be categorized into (as is discussed in [YPZ10a]):

Environment Changes: Web services appear and disappear due to various factors (e.g., network failures, location changes, business changes, etc...). New Web services may be better than the old ones used in the original plans. Therefore, we may consider the use of new Web services and replace the older ones. We model this change as:

$$W' = W - W_{disappear} + W_{appear} \tag{6}$$

Goal Changes: During the execution of the process, business goals may change due to shifting business interests. We assume the removed goals and the new goals are given. We model this change as:

$$g' = g - g_{removed} + g_{new} \tag{7}$$

Fault Caused Changes: A faulty Web service w may also cause the changes. This could be: (1) The faulty Web service has entirely stopped functioning. In this case, it has to be removed from the original plan. (2) The exceptions are due to format

errors. In this case, we can use another Web service to do the format conversion. (3) The exceptions only happen due to certain parameters; the remaining parts of its outputs are still valid. In this case, if we can identify those faulty parameters p, we may project the function of w on the rest of its outputs (i.e. remove p from $out(w)$):

$$out'(w) = out(w) - \{p\} \tag{8}$$

We generally can solve the adaptation problem in two ways: either **re-planning** or **repair**. By **re-planning**, we mean that we try to solve a newly constructed planning problem $P' = ((S', A', \gamma'), s_0, g')$, where A' is an updated set of available Web services, g' is a updated set of goals, with S' and γ' changed accordingly, from scratch. By **repair**, we mean that we try to fix the existing, but broken, plans. We consider that a planning graph is a good candidate for adaptation, because a planning graph models the whole problem world. If the problem changes, part of the planning graph is still valid. In this thesis, we present two repair algorithms (Algorithm 7 and 8) that can grow the partially valid planning graph until a solution is found. It is worth to notice that, although changes can be more Web services available (W_{appear} in Equation 6) or fewer goals needed to be satisfied ($g_{removed}$ in Equation 7), the original planning graph can still give a solution. Therefore, we focus on the cases when Web services are removed or more goals are added, in which cases, the original planning graph becomes only partially valid.

The following sections in this chapter are organized as follows: in Section 3.2, we present a modified composition approach from the previous work [ZY08] (Section 2.4.2) based on a planning graph which represents the re-planning approach to fix broken plan. Next in Section 3.4, we present our repairing approach and its variations. We believe that the repairing approach is better than re-planning in most cases since it can be much faster then the re-planning approach while keeping the similar quality.

3.2 (Re)Planning Algorithm Based on the Planning Graph

As is mentioned in Section 2.4.2, previous work [YZ08] has been done for Web services composition algorithms using planning graphs. Here we apply a backward search

approach to extract solution from a given planning graph. This approach can produce very high quality composition plans in a satisfactory time. The ideas and algorithms can be found in following sections, the implementation can be found at Section 4.6 and the experiments can be found in Section 5.2.

3.2.1 Main Idea

There are mainly two steps in our composition algorithm. The first step is to compose a planning graph from given parameters to goal parameters using the standard planning approach (Algorithm 2 and 3). At each level, the algorithm process invokes all invocable Web services (whose preconditions are satisfied) based on currently known parameters. It stops the expending process when it meets the "Fixedpoint" (Algorithm 4) or all goals are found. Therefore, we can get a planning graph satisfying the given request after this step. The second step is about "extracting" a solution from the constructed planning graph by removing redundant Web services using backward search. This step will give us a lean solution (Definition 10) which has no redundant Web services in it. The details of this process will be discussed in Section 3.2.2.

Definition 10 *Given a solution plan, for any Web service in the plan, if the plan is not valid (Definition 5) after removing a service, we can say the plan is a* ***lean solution****. Otherwise, the plan is a* ***non-lean solution****.*

3.2.2 Backward Search for Extracting Plan

After a planning graph is generated using the standard planning approach we mentioned in previous section (Algorithm 2, 3, and 4), all possible solution paths are contained in the planning graph. We need to "extract" one solution from the planning graph which can have as few services/levels as possible (refer to Section 2.3). Our Backward search provides an approach to extract the optimized solution. The detail of the approach is introduced as follows:

As is shown in Algorithm 6, at each level, the algorithm first calculates a set of subgoals that Web services in the current action level need to fulfill. Next, in line 11, the algorithm calculates a "smallest set" of Web services that can satisfy the subgoals and then substitutes the current action level with the "smallest set" (line 13-15). We use a set named "deferredGoalSet" to maintain the deferred subgoals. This is because

the search process starts backwardly, it's possible that a sub-goal parameter is already produced by a Web service in lower level, and therefore, it is not mandatory for Web services in the current level to satisfy that goal.

As the result, an optimized solution is extracted from the planning graph, which contains only the fewest number of Web services that can satisfy the goals. However, there exists a small chance that the backward search can not find the best optimized solution since it does not compute and compare all possible paths that existed in a given planning graph. It is possible that there exists more than one "smallest set" in a level. The algorithm will pick-up the first one it found and discard the rest. The method to calculate the "smallest set" at each level is discussed in Section 4.7.

Algorithm 6 Backward Search for Extracting Plan

-input: a valid solution plan $G < P_0, A_1, ...A_i, P_i >$
-input: a set of input parameters D_U^{in}
-input: a set of goal parameters D_U^{out}
-output: a optimized lean solution

1: $i = lastLevel$
2: $subGoalSet = D_U^{out}$
3: $deferredGoalSet = \emptyset$
4: **repeat**
5: $subGoalSet \leftarrow deferredGoalSet$
6: $deferredGoalSet = \emptyset$
7: **for** g in $subGoalSet$ **do**
8: compute a subset of services within A_i that can produce g
9: **if** no services can produce g **then**
10: $deferredGoalSet \leftarrow g$
11: $subGoalSet = subGoalSet/deferredGoalSet$
12: $\Gamma = calculateSmallestServiceSet(subGoalSet, i, G)$
13: $A_i = \Gamma$
14: $P_i = \sum(Out(a)|a \in A_i)$
15: $subGoalSet \leftarrow \sum(In(a)|a \in A_i)$
16: $i - -$
17: **until** $i > 0$
18: **return** $G < P_0, A_1, ...A_i, P_i >$

3.3 Repairing versus Re-planning by Example

Let us start with a comparative example. The request is (a,e) and we have nine Web services: A2BC: a→b,c, A2D: a→d, C2E: c→e, D2E:d→e, D2F:d→f, F2G:f→g, F2H:f→h, G2E:g→e and H2I: h→i. The resulting planning graph is shown in Figure 12a and solutions (plans) are A2BC;C2E and A2D;D2E. Let us suppose we commit to the second one. This is the plan we need to repair if anything is broken. We prefer to reuse as much of the original Web services as possible in the new composition, because they are commitments we need to hold. If we remove both C2E and D2E, the graph is broken, because e is no longer produced by any service. This yields a partial planning graph Figure 12b.

If we use re-planning, we simply run the planning algorithm again with a new set of available Web services and a set of updated goals. We then get a new planning graph Figure 13a, from which we can get two new solutions: A2D;D2F;F2G;G2E and A2D;D2F;F2H;H2I. If we instead use the repair algorithm we defined below (Section 3.4), we obtain another graph, Figure 13b, and solution A2D;D2F;F2G;G2E. Our approach is not to build an entire planning graph again, but to rapidly "grow" the partial planning graph in order to obtain a feasible solution. During the growing process, we heuristically search for a direction that can satisfy the broken goals and preconditions, as well as making use of the existing partial graph. We believe repair can be faster than re-planning in most cases, as the partial planning graph is largely valid. Using the repair approach, we can also potentially generate a simpler graph than with re-planning, which means a faster composition time. In addition, the solution we get from repair is of the same quality than with re-planning.

3.4 The Repairing Algorithms Based on Heuristic Search

3.4.1 Motivations

Although the re-planning approach provides a straightforward approach to fix the broken plan when the environment changes, it has several drawbacks:

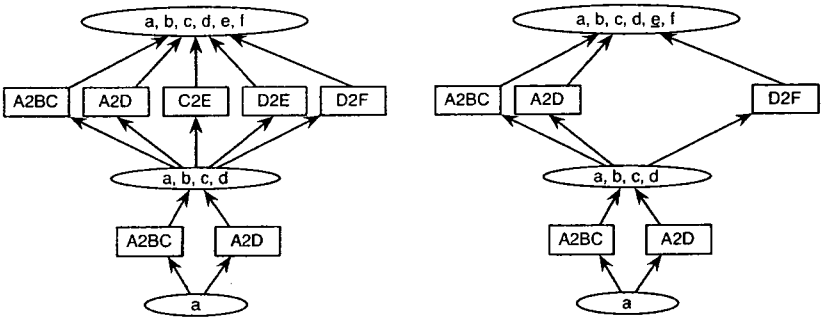

Figure 12: Planning Graph.(Left): original; (Right): after removal of C2E and D2E

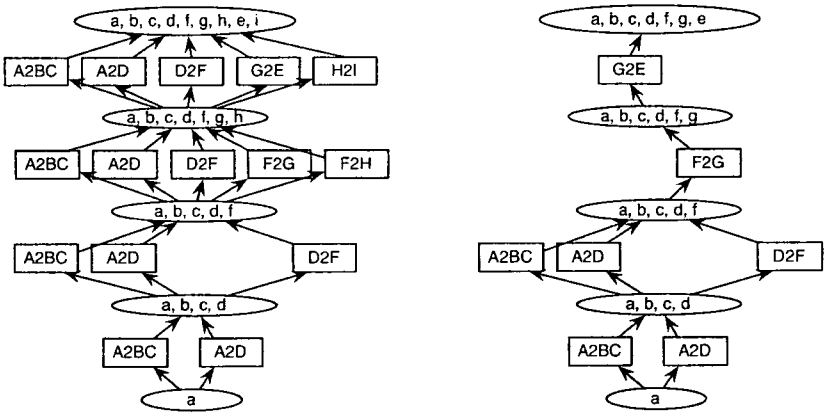

Figure 13: Planning Graph. (Left): by re-planning; (Right): grown by our repair algorithm

- **Time consuming**. Re-planning usually requires similar time costs as the original planning process needed. Thus, it is not very efficient in many cases. For example; if there is only one service in the whole plan that becomes unavailable, re-planning will still need as much time as the original planning process takes.
- **High plan distance** (Definition 7). The re-planning process actually first abandons the whole existing plan. There is no consideration of reusing the existing parts of the plan during its re-planning process. Therefore, the resulting

plan can be "very different" (evaluated by plan distance) from its predecessor. This result will not be acceptable in the real open world sometimes since users already sign contracts with those Web services providers according to its original plan. Changing the entire plan means abandoning all existing business contracts.

Therefore, we need a more efficient mechanism in terms of time and plan distance. Here we propose a heuristic approach based on Greedy search to repair the broken plan. The sections below explain our approach in detail.

3.4.2 Main Idea

When the environment changes, some Web services in the plan might become unavailable and thus need to be removed from the existing plan. Their effects (output parameters) are also removed. This will cause some other Web services which are using those removed parameters as their inputs become "non-invocable". Thus, the first step of our repairing approach is to locate the "broken preconditions" (Definition 11, "BP" for short) in the existing plan. Then, we can evaluate Web services in the updated repository by using our heuristic functions (Section 3.4.3) and insert the best one into the broken position. Notice that when we insert a Web service into broken position to fix BPs the Web service itself may introduce new BPs (i.e. the preconditions of the newly added Web services might not be satisfied by its lower proposition level). Therefore, we need to continue this insertion process until all BPs in the plan have been fixed (i.e. the plan is successfully fixed) or no Web services can fix the BPs (i.e. repairing failed).

Definition 11 *Given a planning graph G and a service w in action level i. $in(w)$ is the preconditions of w. If $in(w) \not\subseteq P_{i-1}$, we say w has* ***broken preconditions*** *and the broken preconditions $= in(w)/P_{i-1}$.*

There are two ways to insert Web services into the broken solutions to fix the BPs:

Create New Levels at the Bottom [YPZ10a]

As is shown in Algorithm 7, the repairing process starts from the highest action level. While $\tilde{g} \neq \emptyset$ (line 2), the repairing process uses the heuristic functions (Equation 9

and 10) to select a candidate Web service w from the Web services repository (line 3). If the w does not exist, the repairing process reports $fail$ (line 4). Otherwise, it adds w to action level n and $out(w)$ to proposition level n (line 5-6). The fixed goals are removed from $\tilde{g}$ (line 7) and the newly introduced broken preconditions are added to BP (line 8). When all $\tilde{g}$ have been fixed, the repairing process moves to lower level (starting from $m = n - 1$, line 10), and continue the insertion process (line 10-18). If BPs still exist after inserting new Web services into the initial action level, the repair algorithm create a new empty action level and insert it right below the previous initial level (line 19-20). The empty action level created will allow the repairing process continue until all BPs are fixed or there exists a BP that no services can fix. Finally, *backwardSearch* is used to extract the solution from the repaired PG (line 29). The heuristic functions are shown in Section 3.4.3. The implementation details are discussed in Section 4.8.

In order to analysis the algorithm complexity, we assume that the processes of adding/removing Web services into action levels costs constant time $O(1)$. Let $|(\tilde{g})|$ be the size of $\tilde{g}$, $|W|$ be the size of available Web services W and $|BP_{all}|$ be all BPs that are required to be fixed. Because the heuristic evaluation process (line 3) checks all Web services in W, and therefore has a complexity of $O(|W|)$, we can deduce that line 1-9 has the complexity of $O(|\tilde{g}| * |W|)$ (worst cases). For line 10-28, the worst cases is the repairing process need to insert $|BP_{all}|$ Web services to fix all BPs (if one Web service only fix one BP), thus it has the complexity of $O(|BP_{all}| * |W|)$. Therefore, the overall complexity (worst case) for above algorithm is $O(|\tilde{g}| * |W|) + O(|BP_{all}| * |W|) + O(backwardSearch)$.

The example is shown in Figure 14. In the example, BP_m is the set containing all BPs existed in level m; the repairing process is trying to fix BPs in BP_m. By applying the heuristic functions, the repairing process selects a Web service "A" and insert it to the action level m (Figure 14b). After the insertion, if $BP_m = \emptyset$, the repairing process moves to the lower level. If it reaches the initial level (assume level $m - 1 = 0$ in Figure 14b) and $BP_{m-1} \neq \emptyset$, the repairing process creates an empty action level below current level (as is shown in Figure 14c). So the repairing process can continue to insert a Web service A' to fix BP'_{m-1}.

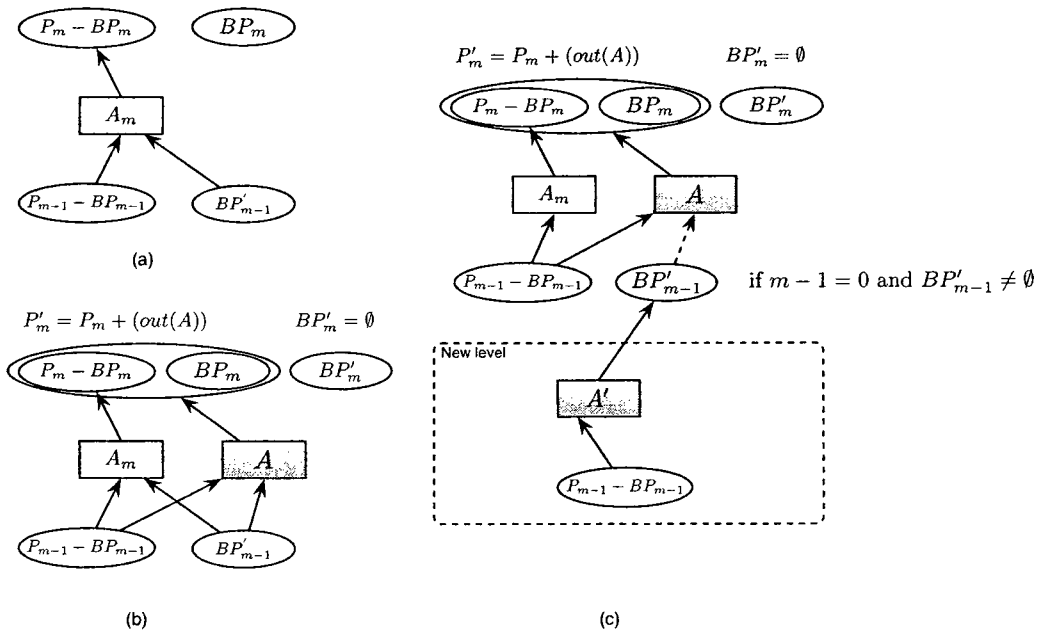

Figure 14: Insert a new level in partial Planning Graph using "Create new levels at the bottom approach"(a) original (b) Insert A to fix BP_m, (c) Insert A' to fix BP_{m-1}

Create New Levels on the Top [YPZ10b]

As is shown in Algorithm 8, the repairing process starts from the highest action level m where the BPs or BGs exists (line 2-5). During the while loop (starting from line 3), we are looking for a set of services A to satisfy BP_m. If there does not exist a set to satisfy BP_i, the algorithm reports $fail$. To look for the set A, we check a set w containing all the services that can produce at least one BP (line 8). From the set w, we apply our heuristic function (Equation 9 and 10) to select one best service to satisfy BP_m (line 9-13). When a service w is added, it can satisfy some of BP_m (line 12). At the same time, we record in C its preconditions that not satisfied by P_{m-1} (line 13). If P_{m-1} can provide all the inputs of the newly added services A (i.e. $C = \emptyset$), the new services are added into A_m level (line 28-29). If not, we need to create a new level $m+1$ (line 15-26). The new action level A_{m+1} can use all the propositions at proposition level P_m. When created, P_{m+1} and BP_{m+1} are assigned as P_m and BP_m respectively. We need to search for a new set A to satisfy BP_{m+1} (line

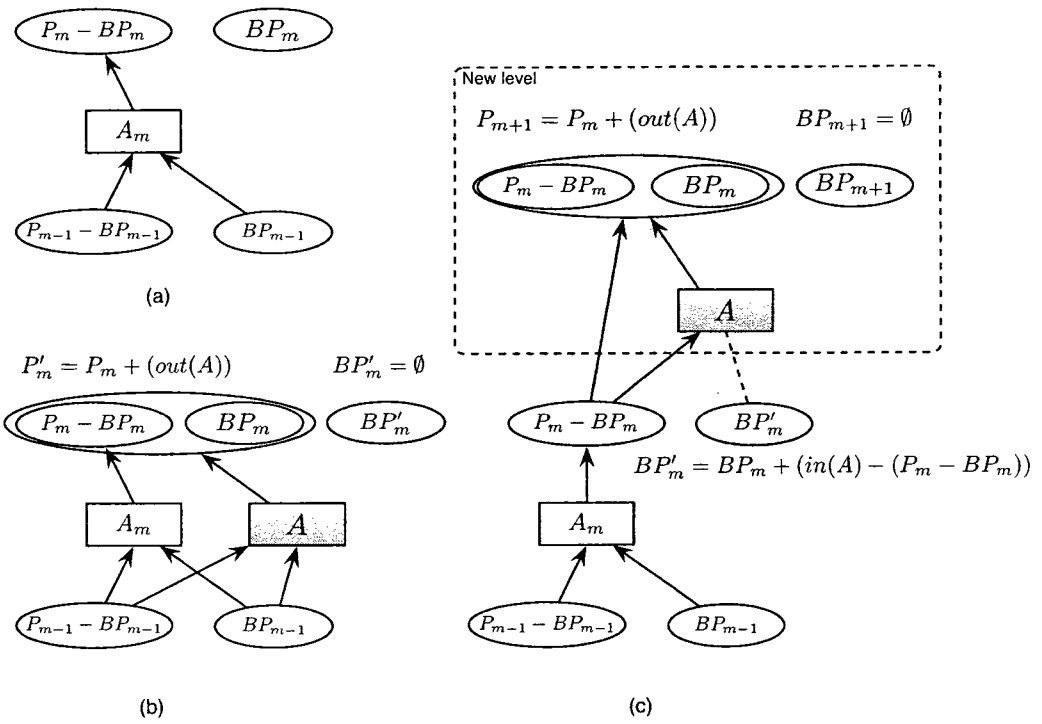

Figure 15: Insert a new level in partial Planning Graph using "Create new levels on the top approach"(a) original (b) A can be satisfied by P_{m-1}, (c) add a new level for A

18-21). If we succeed, we can update P_{m+1}, A_{m+1} and BP_m (line 23-25), otherwise fail (line 22). Notice that the unsatisfied precondition of A is added to BP_m. We repeat this process until all BP_i become empty. If all BP_i are empty, we succeed to repair the planning graph. The repaired planning graph is not a "full" planning graph, but contains at least one solution. The $backwardsearch(G)$ is a regular backward search function as in graph planning technique to get the final plan. For graph planning, the backward search is the most expensive step. As we construct a smaller graph for this search than a "full" planning graph generated from re-planning. It is expected that we will get a solution faster.

Similar to the algorithm complexity we discussed in above section for Algorithm 7, we assume that the processes of adding/removing Web services into action levels costs

constant time $O(1)$. Let $|(\tilde{g})|$ be the size of $\tilde{g}$, $|W|$ be the size of available Web services W and $|BP_{all}|$ be all BPs that are required to be fixed. The heuristic evaluation process only checks a subset V of all Web services W such that every Web service in V can produce at least one broken precondition in BP, and therefore has a complexity of $O(|V|)$. For line 2-3, the worst case is $\tilde{g} \neq \emptyset$, and therefore it has the complexity of $O(|\tilde{g}| * |V|)$. For line 6-28, the worst case is it need to loop $|BP_{all}|$ times to fix all BPs, and therefore it has the complexity of $O(|BP_{all}| * |V|)$. Hence, the overall complexity for the algorithm is $O(|\tilde{g}| * |V|) + O(|BP_{all}| * |V|) + O(backwardSearch)$.

The example is shown in Figure 15. It shows two situations after using the heuristic function to select a Web service "A": (1) If the preconditions of Web service "A" can be completely satisfied by P_{m-1}, the repairing process will insert "A" to current action level as is shown in Figure 15b; (2) If the preconditions of Web service "A" cannot be completely satisfied by P_{m-1}, the repairing process will create a new level right below current level (i.e. level $m + 1$) by cloning current level (as is shown in Figure 15c) and then insert "A" to the newly created action level.

The detail algorithm is shown in Algorithm 8. The heuristic functions are shown in Section 3.4.3. The implementation details are discussed in Section 4.8.

3.4.3 Heuristic Function

There are two heuristic functions we used in the algorithm. The first one is for inserting a service into the highest action level, in which case there are no broken preconditions only the broken goals. The second one is for inserting a service into any action level lower than the highest one. The equations are as follows: Assume:

- w: a service whose inputs are $in(w)$ and outputs are $out(w)$
- G: a partial planning graph
- $\tilde{g}$: a set of unimplemented goals
- BP: a set of unsatisfied preconditions of some services in G

If we want to add an action w to the highest action level n, the evaluation function is:

Algorithm 7 Search for a repair plan (create new levels at the bottom): $Repair1(W, G, BP, \tilde{g})$

-Input: available service set W

-Input: partial planning graph G

-Input: set of broken precondition BP

-Input: set of unimplemented goals $\tilde{g}$

-Output: either a plan for $fail$

result = $fail$
while $\tilde{g} \neq \emptyset$ **do**
  select an action w with the best $f(G, w)$ according to Equation 9
  if w does not exist **then** break
  add w to G at action level A_n and $out(w)$ at proposition level P_n
  remove $\tilde{g} \cap out(w)$ from $\tilde{g}$
  add $in(w) - P_{n-1}$ to BP
if $\tilde{g} \neq \emptyset$ **then return** result
for $m = n - 1; m > 0; m - -$ **do**
  while $BP \cap P_m \neq \emptyset$ **do**
    select an action w with the best $f(G, w)$ according to Equation 10
    if w does not exist **then** break
    add w to G at action level A_m and $out(w)$ at proposition level P_m
    remove $P_m \cap BP \cap out(w)$ from BP
    add $in(w) - P_{m-1}$ to BP
  if $BP \cap P_m \neq \emptyset$ **then return** result
while $BP \neq \emptyset$ **do**
  insert an empty proposition level P_1 and empty action level A_1
  $P_1 = P_0 - BP$
  select an action w with the best $f(G, w)$ according to Equation 10
  if w does not exist **then** break
  add w to G at action level P_1 and $out(w)$ at proposition level P_1
  remove $P_1 \cap BP \cap out(w)$ from BP
  add $in(w) - P_0$ to BP
if $BP \neq \emptyset$ **then return** result
result = $backwardsearch(G)$
return result

Algorithm 8 Search for repair plan (create new levels on the top): $Repair2(W, G, BP, \tilde{g})$

-Input: available service set W

-Input: partial planning graph G

-Input: set of broken precondition BP

-Input: set of unimplemented goals $\tilde{g}$

-Output: either a plan for $fail$

1: result = $fail$
2: **if** $\tilde{g} \neq \emptyset$ **then**
3: $m = n, BP_m \leftarrow \tilde{g}$
4: **else**
5: $m = max(i)$, where $BP_i \neq \emptyset, i \in [1:n]$
6: **while** $BP_m \neq \emptyset$ **do**
7: $A = \emptyset, B = BP_m, C = \emptyset$
8: $V = \{w | w \in W, out(w) \cap BP_m \neq \emptyset\}$
9: **while** $BP_m \neq \emptyset$ **do**
10: select an action $w \in V$ with the best $f(G, w)$ according to Equation 10
11: **if** w does not exist **then** break
12: $BP_m = BP_m - out(w), A = A + \{w\}$
13: $C = C + in(w) - (P_{m-1} - BP_{m-1})$
14: **if** $BP_m \neq \emptyset$ **then** break
15: **if** $C \neq \emptyset$ **then**
16: insert a new action level $m+1$ with $A_{m+1} = \emptyset$, $P_{m+1} = P_m$, and $BP_{m+1} = BP_m = B$
17: $A = \emptyset$
18: **while** $BP_m \neq \emptyset$ **do**
19: select an action $w \in V$ with the best $f(G, w)$ according to Equation 10
20: **if** w does not exist **then** break
21: $BP_m = BP_m - out(w), A = A + \{w\}$
22: **if** $BP_m \neq \emptyset$ **then** break
23: $BP_{m+1} = \emptyset$
24: $P_{m+1} = P_m - B + \sum\{out(w) | \forall w \in A\}$
25: $BP_m = \emptyset$
26: $A_m = A_m + A$
27: $m = max(i)$, where $BP_i \neq \emptyset$, $i \in [1:n]$
28: **if** $BP_m \neq \emptyset$ **then** return result
29: $result = backwardsearch(G)$
30: return result

Equation for broken goals

$$f(G,w) = |\tilde{g} \cap out(w)| * 10 + |P_{n-1} \cap in(w)| - |in(w) - P_{n-1}| - e(w,G) \quad (9)$$

Where $|\tilde{g} \cap out(w)|$ is the number of unimplemented goals that can be implemented by w and the coefficient 10 is the weight of this term. It shows that it is more important to satisfy the goals than the other needs represented by the following terms; $|P_{n-1} \cap in(w)|$ is the number of inputs of w that can be provided by the known parameters at the level P_{n-1}; $|in(w) - P_{n-1}|$ is the number of inputs of w that **cannot** be provided by the known parameters at level P_{n-1}. This set needs to be added into BP, if w is added. Finally, $e(w,G)$ is the number of the actions in G that are exclusive to w.

If we want to add an action w to action level m, and m is not the goal level, the evaluation function is:

Equation for broken preconditions

$$f(G,w) = (|\tilde{g} \cap out(w)| + |\sum_{i \geq m} (BP_i \cap out(w))|) * 10 + |P_{m-1} \cap in(w)| - |in(w) - P_{m-1}| - e(w,G) \quad (10)$$

Compared to Equation 4, the above equation added term $|\sum_{i \geq m} (BP_i \cap out(w))|$ which is the number of broken propositions in and above level m that can be satisfied by the outputs of w.

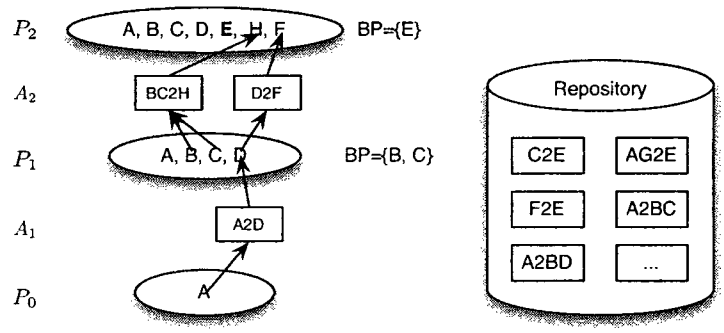

Figure 16: Example of Heuristic Functions

Example

As is shown in Figure 16, the partial planning graph has $BP = \{E\}$ for P_2 and $BP = \{B, C\}$ for P_1, E is the goal. Using the repairing algorithm 7 for example, the repairing process starts from the highest action level A_2. In order to fix $BP = \{E\}$, it first searches the service repository and selects all Web services that can produce E as the candidate Web services, in this case it gets a candidate list $\{C2E, AG2E, F2E\}$. Then, it uses the heuristic function (Equation 9) to evaluate each of the candidate Web services, and it gets: $f(G, w_{C2E}) = 11, f(G, w_{AG2E}) = 10$ and $f(G, w_{F2E}) = 9$. $C2E$ has the highest heuristic and therefore is inserted to A_2. Next, in order to fix the $BP = \{B, C\}$ for P_1, the repairing process repeats the process. It first selects a list of candidate Web services that can produce at least one of BP: $\{A2BC, A2BD\}$. Then, it uses the heuristic function (Equation 10) to evaluate each of the candidate Web services, it gets: $f(G, w_{A2BC}) = 21$ and $f(G, w_{A2BD}) = 11$. Therefore, $A2BC$ is inserted to A_1 to fix the BP.

3.4.4 Indexing for Repairing Algorithm

There are two situations when an exiting solution breaks: (1) each broken Web services in the solution can be fixed by a replacement service that is still available. We call this "replaceable situation"; (2) there exists a broken Web services that doesn't have any replacement Web services, or all of its replacement Web services are unavailable (e.g. broken as well). In this case, the preconditions of some other Web services which depend on the outputs of this broken Web services will never be satisfied by any other Web services (note: the discussion here is based on the WSC dataset, in which solutions are independent to each other (i.e. two solutions do not share any of Web services). We call this "non-replaceable situation". The heuristic repairing algorithm will likely become trapped in this situation since it tries to fix some BPs that are not fixable. In order to avoid this kind of trap, we propose an indexing approach which shows as follows:

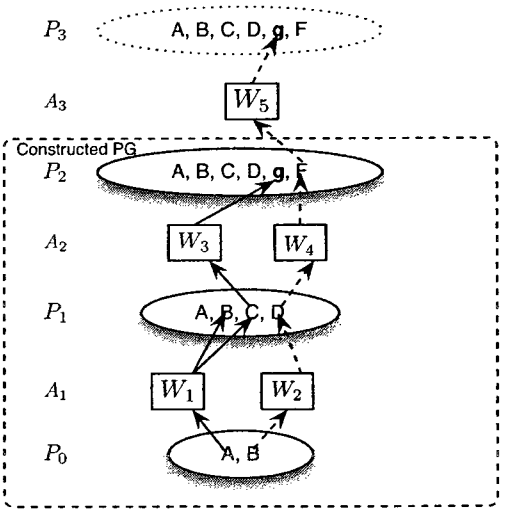

Figure 17: Example of Indexing Approach

Before Solution Breaks (Indexing):

1. Generate a Planning Graph (PG) which may contain multiple solutions (including the current solution). We name the resulted Planning Graph as "Constructed PG". As the example shown in Figure 17, the "Constructed PG" contains a solution $< W_1, W_3 >$ and parts of another solution $< W_2, W_4, W_5 >$.

2. Index the existing solution that is extracted from the "Constructed PG" by backward search. In the example, we index the existing solution $< W_1, W_3 >$.

3. For each service in the existing solution, index their replacement Web services (Definition 12). As is shown in Table 1, W_1 has three replacement Web services: W_{11}, W_{12}, W_{13} and W_3 does not have any replacement Web services.

Services	Replacements
W_1	W_{11}, W_{12}, W_{13}
W_3	

Table 1: Index Table of Replacement Web Services

After Solution Breaks (Repairing):

1. For each of broken Web services, check their indexes of replacement Web services. If there is an available replacement service (i.e. not broken), use it to replace the broken Web services. For example, if W_1 breaks, we can simple use W_{11} to replace it by looking up the index table. If there exists a broken Web services that doesn't have any replacement Web services or all of its replacement Web services are unavailable, remove all Web services in the current solution as well as their replacement Web services from the indexed "Constructed PG". For example, if W_3 breaks, we have to remove all $\{W_1, W_3\}$ as well as their replacements $\{W_{11}, W_{12}, W_{13}\}$ from the indexed "Constructed PG".

2. Use heuristic repairing algorithms (Algorithm 7 and 8) to fix all BPs and BGs in the "Constructed PG". In this example, when we completely removed the Web services of solution 1 ($\{W_1, W_3, W_{11}, W_{12}, W_{13}\}$), we have broken goal g need to be fixed. By applying our repairing algorithms, it finds a Web service W_5 that can produce g and therefore fix the partial planning graph.

3. If the fixing process succeeds, use "backward search algorithm" to extract one solution. In this example, we get the solution: $< W_2, W_4, W_5 >$.

Definition 12 *Given a lean solution:* $PG =< P_0, A_1, P_1, A_2, P_2 ... A_i, P_i >$ *and a service* $a1 \in A_m$. *We define a function* $usedOutputs(a1) = output(a1) \cap \bigcup \{input(A_{m+1}), ..., input(A_n)\}$, *which represents the subset of service* $a1$*'s outputs that is used by Web services in its higher levels as their inputs. If there exists another service* $a2$ *while* $input(a2) \subseteq input(a1)$ *and* $output(a2) \supseteq usedOutputs(a1)$, *we say* $a2$ *is an replacement service to* $a1$ *for the solution PG.*

The indexing approach mentioned above allows repairing processes to identify whether we are facing a "replaceable" or "non-replaceable" situation in an extremely short time. If replacement Web services cannot be found we need to remove all necessary Web services to prevent our heuristic algorithm getting trapped. Also, because this approach lets the repairing process repair BPs on the "Constructed PG", it ensures the reuse of the existing solution parts. Repairing process reuses those solution parts and "grows" the remaining parts using heuristics. Experiments for this indexing approach can be found in Section 5.4.

Chapter 4

Implementation

4.1 Implement the Web Service Composition Test Bed

We have implemented the algorithms discussed in Chapter 3 in Java. We also use the platform which is initially developed for the Web Service Challenge 2009 competition to generate datasets and perform evaluation (the platform will be introduced in detail in Section 4.3.1). In this section, we are going to discuss the details of our implementation. (also cf. Appendix A)

4.2 Concepts, Things, Services and Parameters

Before we discuss about the implementation detail, it is very important to understand the following terms which will be used extensively throughout this chapter [Ble10].

- Concept - Defined in OWL (Web Ontology Language). Concept stands for a group of "things" that share common characteristics (or attributes). A concept can be subsumed by the other concept, e.g., a concept "ISBN" subsumes "ISBN-8" and "ISBN-16".
- Thing - Defined in OWL (Web Ontology Language). Things are concrete instances of concepts. Things belong to the same concept which shares the same

set of attributes but might have different value for those attributes, e.g., "H1L 235" and "H1L 236" are two instances of the concept "Post Code".

- Params - Parameters are message parts defined in WSDL (Web Service Definition Language). A parameter can be associated with a semantic concept defined in OWL. It accepts all instances of the associated concepts as well as all instances of the concepts that are subsumed by its associated concepts.

- Services - Services 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.

4.3 Experiment Setup

4.3.1 Dataset Generation

We use the Web Service Challenge 2009 platform and its tools in our experiments [Ble10]. This platform contains a challenge client, a dataset generator (Figure 18) 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 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. To generate a dataset, user needs to specify the number of services the dataset will have, the number of solutions and their length (in steps). Given those parameters, the generator randomly generates a set of given concepts and goal concepts first. 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 provides those inputs/outputs. Then, based on the solutions, the dataset generator generates the complete ontology and Web service interface set by padding new concepts and services which are not used in the solutions.

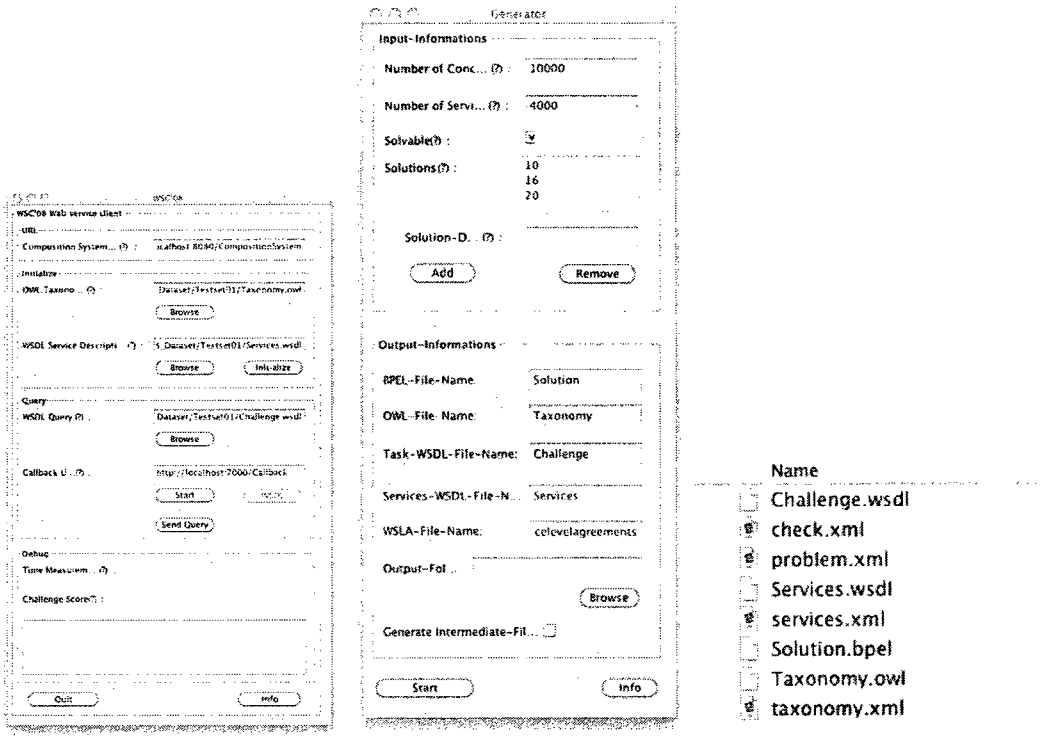

Figure 18: Challenge Client (Left) and Data Set Generator GUI (Middle) and Date Set Files Generated (Right)

As Figure 18 shown, the generated dataset contains several files including "Services.wsdl", "Taxonomy.owl", "Challenge.wsdl" and "Solution.bpel". They are the essential files that are necessary to run our experiments:

- "Services.wsdl" - It is a big file containing services description of all Web services generated (in WSDL). A "Services.wsdl" generated in one of our experiments contains more than 4000 Web services. Our algorithms will parse these WSDL documents to create the service repositories which can be used in composition processes. Notice that each Web service in "Services.wsdl" has exactly one "port type" and each "port type" has one "input message" and one "output message".

- "Taxonomy.owl" - This file contains all semantic "concepts" (classes) and "things" (instances) that are associated to input and output parameters of generated Web

services. Concept stands for a group of "things" that share common characteristics (or attributes). A concept can be subsumed by the other concept. For example, a concept "ISBN" subsumes "ISBN-8" and "ISBN-16". "Thing" are concrete instances of concepts. Things belong to the same concept which shares the same set of attributes but might have different values for those attributes. For example, "H1L 235" and "H1L 236" are two instances of the concept "Post Code". Our algorithm will parse this file to understand the semantic relationship between Web service message parameters.

- "Challenge.wsdl" - This file represents a desired composite Web service. 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. For example, the input parameters could be "current address" and "food preference" and the desired parameter could be "Nearest Restaurant that has the preferred food".

- "Solution.bpel" - This file contains all possible "lean solutions" that exists in the generated dataset. It can be used to validate the correctness of the solutions given by our algorithm and compare their quality.

4.3.2 Algorithms Invocation Procedures

As is mentioned in Section 4.3.1, we use the platform and tools provided by Web Service Challenge 2009 to test the performance of our algorithms. We also follow the competition rules that are specified by the Challenge: As Figure 19 shows, the composition system contains two parts: the client(Right) and the server(Left). Our algorithm resides in the server part as a Web service. The client is also a Web service so that it can communicate with the server. In order to run our algorithms:

1. Use the dataset generator provided by WSC to generate a dataset using specified parameters (number of services, solution depths, etc...) and upload the dataset to the client.

2. Once the client has the dataset, we can click the "Initialize" button to ask the client to send two hyperlinks which point to "Services.wsdl" and "Taxonomy.owl" to the server.

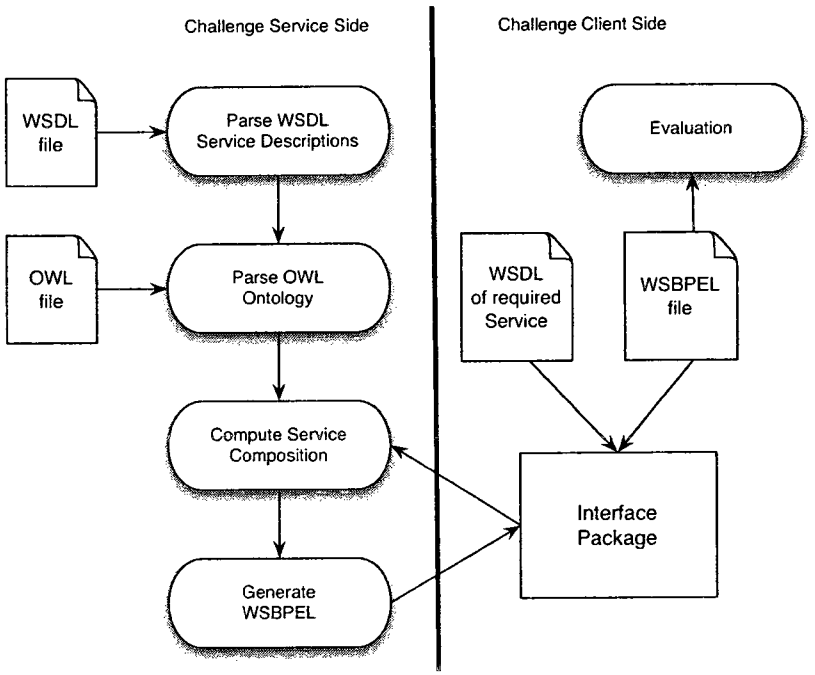

Figure 19: The procedure of the Web Service Challenge [Ble10]

3. The server then parses the above two files to perform necessary initialization processes (parsing, modelling, indexing, etc...). The server will send a message to notify the client once the process is completed.

4. Once the initialization is over, we can click "Query" button in the client GUI to send out composition queries to the server (in form of "Challenge.wsdl") and start the timer.

5. The server parses queries, performs calculation using our algorithm, and sends the solution back to the client (in the form of BPEL document).

6. Upon the receive of BPEL document from server, the client stops the timer and validates the correctness of the given solution by the server. The validation result as well as composition time then show up on the client.

4.4 Building Models

Since we use Java programming language for our implementation, we would like to take its benefits by modelling the composition problem in Object-Oriented way. As Figure 20 shows, we have modelled the problem using six objects. At the top, there is an abstract object, named "UniNameObject", which represents an object with unique name. It also contains many useful methods such as sorting, comparing, etc... which can be inherited by its sub-classes. "Concept", "Service", "Thing" and "Param" inherit "UniNameObject". Besides the common attributes and methods they inherited, they are also built in with index tables which will help to expedite the composition process (Indexing details refer to Section 4.5.1). For example, in each "Concept" object, it maintains a set of services which accepts this concept as inputs. Also, it maintains a set of concepts that subsumes this concept and a set of concepts which is subsumed by this concept. The object "PlanningGraph" represents a Planning Graph which contains an action level array (storing services invoked at each level) and a proposition level array (storing concepts generated at each level). The implementations of all algorithms discussed in this thesis are based on these model objects.

4.5 Implementation Details

4.5.1 Indexing the Data

We apply several indexing techniques for expediting composition processes. We first parse the given WSDL file and OWL file into our model objects mentioned above. By doing this, all necessary information for running the algorithm stays in the memory. So it does not need to search though XML file every time. Second, for each concept, we index its super classes and sub classes. Therefore, the algorithm does not need to re-compute this every time when checking semantic subsumption. Third, for each service, we use a hash table to index all concepts (defined in a OWL document) that the service takes as inputs or outputs. The subsumption hierarchy is "flattened" in this step so we do not need to consider semantic subsumption during the planning processes (Section 4.5.2). Last, we also use a hash table to store the mapping relationships between each semantic concept and all services that can accept that

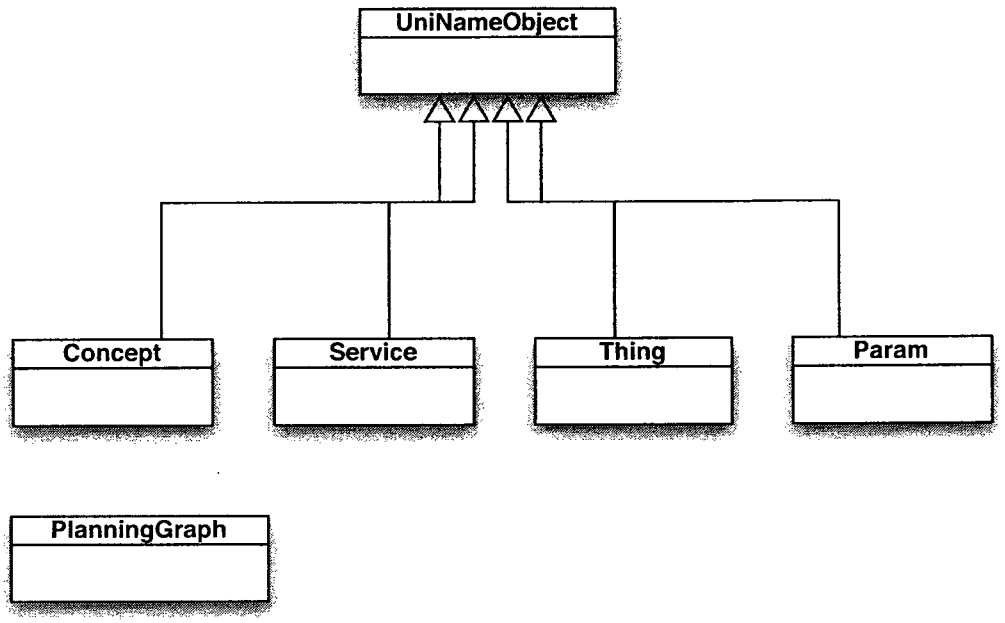

Figure 20: Class Diagram for Model Objects

concept as one of their inputs (see Table 2). This is similar to the "reverse indexing approach" introduced in [YXG08]. It allows us to search the invokable Web services from the concepts quickly - a simple join operation among related rows of the index table instead of checking the precondition of each service in the repository. Table 2 shows an example of the reversed index table. For example, if we currently know $\{$"$concept1$", "$concept2$"$\}$, we can get currently invokable services by looking up the reversed index table and join their associated services sets. In this case, it would be the intersection of $\{serviceA, serviceB\}$ and $\{serviceC, serviceD, serviceE\}$.

4.5.2 Flatten the Semantic Relationships

As Figure 21 shown, the semantic relationships among things and concepts are defined in the OWL document. We need to check these relationships to know whether a parameter can be accepted by a Web service as its input. In this example, we have four concepts. At the top level, we have a concept, named "Machine", which subsumes

Concept	Services that accept this concept
concept1	serviceA, serviceB
concept2	serviceC, serviceD, serviceE
concept3	
concept4	serviceF
...	...

Table 2: Example of Reverse Index Table

the concept "Vehicle", and concept "Vehicle" subsumes two concepts named "Car" and "Motorcycle". We also see two Web services named "A" and "B". Web service A has an output "Ford 1986 Red" which is an instance of "Car" and Web service B accepts an input "An old Vehicle" which is an instance of "Vehicle". By checking the semantic relationships, we can know that the output of Web service A can be accepted by Web service B because "Car" is also one kind of "Vehicle".

In order to composite the planning graph, we need to get a list of currently invokable Web services as candidates based on currently known parameters at each level. Without knowing the semantic relationship between their I/O parameters in advance, we have to check the relationship map in OWL every time, which is extremely time consuming. In order to expedite this process, we apply an approach to flatten the semantic relationships by converting them into syntactic problems. For a Web service:

- For each of its output parameters, we calculate its directly associated concepts as well as all concepts that subsumes the concept. We index the resulted concept set as all concepts that Web services can produce. For example, Web service A can produce "Car", "Vehicle" and "Machine".

- For each of its input parameters, we only calculate its directly associated concepts. We then index the resulted concept set as all concepts that the Web service can accept. In above example, the acceptable concept set of Web service B is "Vehicle".

- By comparing the I/O concept set of Web services A and B, we clearly see Web service B is invokable after we invoke Web service A because "Vehicle" will be provide by invoking Web service A. No semantic relationship checking

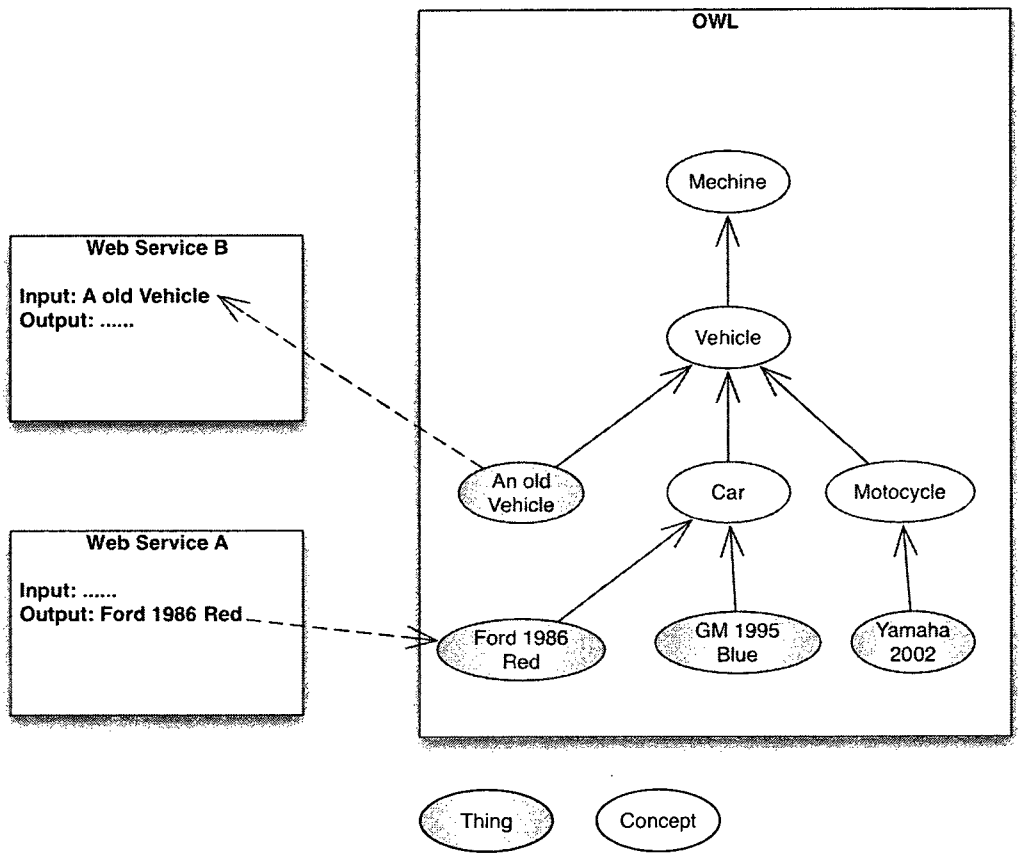

Figure 21: Semantic Relationship Between Web Service I/O parameters

using OWL is necessary in this approach. Thus, the efficiency of the whole composition process improves.

4.6 Planning Graph Construction

As is Section 3.2 introduced, the planning algorithm generates a Planning Graph from given known parameters (which means "concept" in our implementation since we already "flatten" the semantic relationship) to find desired goal parameters. It maintains a set of currently known concepts and, based on which it continues to invoke all possible Web services whose preconditions are satisfied by currently known concepts. The loop continues until it reaches the "Fixedpoint" [GNT04] or all goals

have been found. "Fixedpoint" is a state that if the algorithm continues to expend the planning graph by invoking services, the proposition will not change. Appendix B shows the complete Java source code for this algorithm implementation.

4.7 Backward Search Implementation

The most crucial part of the backward search process implementation is to implement the "smallest sets" of Web services that could satisfy subgoals at each level. By calculating the "smallest set" at each level, we can extract a optimized solution from the planning graph. The details of the approach is discussed as follows:

Calculating Smallest Sets of Web Services to Satisfy Subgoals

Given a set of Web services W and a set of subgoals G, the "**smallest set**" of W to satisfy G is the smallest subset of W such that every subgoal in G can be produced by at least one Web service in W. Our backward search process continues to calculate the "smallest sets" of Web services in each action level which can satisfy the subgoals. In order to illustrate the concept, we present an example here as is shown in Figure 22.

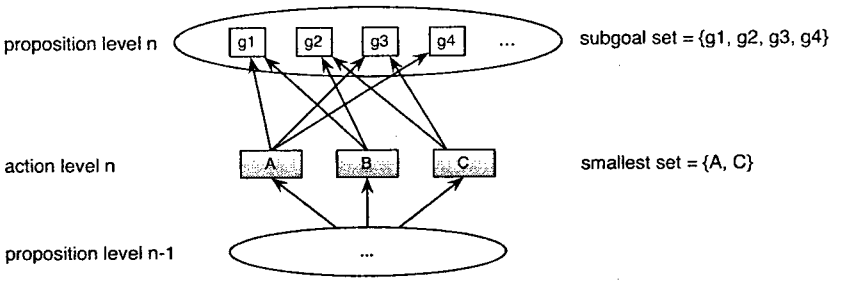

Figure 22: An Example of the Smallest Set

Subgoals	subgoal g1	subgoal g2	subgoal g3	subgoal g4
Web services	A, B	C, B	A, C	A

Table 3: The Originated Web Services for Subgoals

In this example, We have three Web services "A", "B" and "C". As is shown in the figure, Web service "A" has the outputs {g1, g2, g3, ...}; Web service "B" has the outputs {g1, g2, ...} and Web service "C" has the outputs {g2, g3, ...}. Therefore, we can have the Table 3 which shows the set of originated Web services for each subgoal in the proposition level n. From the table, we can easily see that only two sets of Web services, {A, B, C} and {A, C}, can satisfy the condition "every subgoal in G can be produced by at least one Web service in W" that we mentioned above. The set {A, C} has fewer number of Web services and therefore is the "smallest set" that can satisfy the subgoals. The detail approach to calculate the "smallest set" is shown as follows:

1. We need to calculate a set of subgoals that are required to be satisfied by Web services in the current action level. As is shown in the the example, the resulted subgoal set is $\{g1, g2, g3, g4\}$ in the current level;

2. We build the "subgoal-to-origins" relationships as is shown in Table 3. In the example, we have a subgoal-to-origins relationships $\{g1 : \{A, B\}, g2 : \{C, B\}, g3 : \{A, C\}, g4 : \{A\}\}$;

3. With "subgoal-to-origins" relationship built, we then let the algorithm calculate all subsets of Web services in the current action level. In the example, the current action level contains Web services $\{A, B, C\}$, so we can get 8 subsets $\{A\}, \{B\}, \{C\}, \{A, B\}, \{B, C\}, \{A, C\}, \{A, B, C\}$ and $\emptyset$.

4. For each subsets, we try to remove it from the "subgoal-to-origins" relationships (i.e. $\{g1 : \{A, B\}, g2 : \{C, B\}, g3 : \{A, C\}, g4 : \{A\}\}$) and check if all subgoals can still be produced by at least one Web service. If yes, we add the subset to the "smallest set" candidate list.

5. We compare the size of all candidate subsets and pick up the one has the smallest size. That set is the "smallest set" to satisfy the subgoals in current level. Note that it is possible that there are more than one "smallest set" existed in one level. In that case, our algorithm will pick up the one it first found.

Appendix C shows the main part of the backward search algorithm implementation. The sample execution log can be found in Appendix F. Also, the complete source code is available for checkout online, cf. Appendix C

4.8 Repairing Algorithm Implementation

As is Section 3.4 introduced, when an existing plan is damaged, we have two options to fix it: first one is re-planning which discards the whole plan and re-compose the whole plan. If the damaged part is relatively small (e.g. only one service in the whole plan become unavailable) using re-planning approach might not be very efficient. Therefore, we propose a repairing approach which can locate the damage part and fix it very quickly. There are mainly four parts of the implementation:

Removing broken Web services: as the precondition of the repairing process, a set of removed Web services is generated (depends on different experiments). Therefore, we need to first remove them from the given solution plan and mark the broken preconditions of removed Web services;

Repairing broken plan: according to our repairing algorithms (Algorithm 7 and 8) the repairing process repeatedly use the heuristic functions (Equation 9 and 10) to evaluate candidate Web services and insert them into the plan. In our implementation, in order to improve the performance, we select the Web services which can produce at least one goal as the candidates. Therefore, the repairing process does not need to apply heuristic functions to all available Web services in the repository;

Extracting solution: after repairing process succeed, we have a repaired solution. However, the solution might not be lean solution. Therefore, we use our backward search algorithm (Algorithm 6) to extract the optimized solution from the solution PG;

Validating solution: a solution validator (Section 4.8.1) is implemented to validate the correctness of resulted plan and calculate the plan distance (Definition 7) from its original solution plan.

As an example, the Appendix E shows the Java code of one repairing algorithm (Algorithm 7).

4.8.1 Plan Validator

Validating the correctness of our algorithm results is the essential part of our experiments. The results of our experiments represent by planning graphs, and therefore we need a plan validator which can tell whether a given planning graph is valid or not (Definition 5). The aspects checked by our planning validator are shown as follows:

Aspects Checked by the Plan Validator

1. The initial proposition level. It will check whether the initial proposition level only contains the initial given concepts.

2. The last proposition level. It will check whether the last proposition level contains all desired goal concepts.

3. Invocable services. It will loop through each action level in the plan (start from level 1) to check if its previous proposition level satisfies the precondition of each service in the current action level.

4. Proposition level correctness. It will loop through each proposition level in the plan (start from level 1) to check if the concepts in this proposition level is exactly equal to the intersection of current known concepts set and all concept produced by its corresponding action level.

Notice that the plan validator we developed is different from the "solution checker" that is provided by WSC2009 Platform. The "solution checker" checks the composition results in BPEL format while our plan validator checks the composition results in planning graphs. It is possible that we convert our experiment results from planning graphs to BPEL-documents, and then use the "solution checker" to do the double-check.

4.8.2 Removing Services from Existing Plans

The precondition of this removing process is that we have a list of services that become unavailable. Then removing process uses the list to check whether there are any services affected in the given plan first. If there is no service affected then it will return the plan since no removing or repairing is necessary. However, if there

are some services affected by the change, it will first try to remove these affected services as well as the concepts they produced from the plan (if these concepts does not produced by any other service). Then, it will use plan validator to check if the plan is still valid after such removal, if yes it goes to repairing process otherwise it just return the resulted plan.

4.8.3 Heuristic Evaluator

Heuristic Evaluator is able to calculate the heuristic score of a given service according to the heuristic function. We implement several heuristic functions as is Section 3.4.3 mentioned, they can be used in different context. For example, a simple heuristic function implementation for backward repairing process is showed in Appendix D. Notice that we apply a weight (10) to each of the goal concept that the service can produce, so that the evaluation can be more accurate.

4.8.4 Repairing Based on Heuristic Search

We implemented both varieties of the repairing algorithms (Algorithm 7 and 8) shown in Section 3.4. Here we use the Algorithm 7 as an example to illustrate our repairing algorithm implementation:

As the Algorithm 7 shown, there are mainly two phases for the repairing process:

Step 1: starting from last action level, backwardly, the repairing process continue to insert Web services to fix broken preconditions at each existing level;

Step 2: when the repairing process reaches the initial action (i.e. action level 1), if there are still broken preconditions needed to fix, the algorithm needs to create and insert a new empty action level and its corresponding preposition level to level 1, so that the repairing process (i.e. Step 1) can continue until the plan is fixed or repair failed.

The detail steps from the implementation perspective are described as follows:

Start from the highest Action Level ($n = highestlevel$). While level counter is greater than 0 and the BPs (broken preconditions) list is not empty, continue the loop:

1. Compute a "current sub-goal set" which contains all BPs that need to be satisfied right in current propositional level (level n). Initially, this set will contains all unsatisfied goals only;

2. Select all Web services that are not already in the current action level and could produce at least one sub-goal in "current sub-goal set" as candidates. Compute heuristic score of all candidates based on the heuristic function;

3. Insert the Web service that has the highest score in candidate list to current action level. Add its outputs to the proposition level n and its inputs (i.e. preconditions) to the proposition level $n-1$. Check if level n still has unsatisfied subgoals. if yes, repeat step 1 unless no candidate can be found, in which case, it should return "repair failed";

4. Check if all BPs haven been fixed, if so, return PG;

5. Decrease level counter by 1 (i.e. $n = n - 1$)

6. If it reaches the initial level (i.e. $n = 0$), insert a new empty action level and its corresponding propositional level below the current level. Set level counter $n = 1$. Continue the repair process.

The repairing process will report failure if no candidate services can fix the remaining broken preconditions. Appendix E shows the repairing implementation in Java source code.

It is worth to notice that, the Algorithm 8 we discussed in Section 3.4 has the similar implementation as the one we illustrate above (Algorithm 7). The main difference is, before moving the level counter to a lower level (i.e. $n = n - 1$), the repairing processes of the Algorithm 8 make sure there are no BPs exist in its current level and the levels above (i.e. $BPs = \emptyset$ for $levels >= n$), while the repairing processes of the Algorithm 7 will move to lower level regardless of whether or not BPs exists in its higher levels. It is expected that the Algorithm 8 can have better

performance in terms of time since it reuse the existing parts of the plan during its repairing processes.

Chapter 5

Experiments

5.1 Datasets Used in Experiments

In order to compare the results in different experiments , we pre-generated three datasets as is shown in Table 4. All our experiments in this section are based on these three datasets. The dataset 1 has relatively small number of Web services. It contains solutions in four different depths (i.e. solutions can be found in 9, 18, 19 or 27 levels). The dataset 2 has relatively large number of Web services and its solutions have 3 different depths. The dataset 3 has similar number of Web services as dataset 2 but its solutions only have one depth. In addition, The best solutions for each dataset are also known from the WSC Dataset Generator; for example, the best solution of dataset 1 contains 10 Web services and 9 levels.

As a naming convention we use in this section, the solutions that have the same depth in a dataset are referred to as "solution n". For example, we can simply say that in dataset 1 there are 4 solutions, solution 1 has 9 levels, solution 2 has 18 levels, solution 3 has 19 levels and solution 4 has 27 levels.

There is one very significant characteristic of the generated datasets needed to be mentioned here: each dataset can have solutions in different depths. Solutions in different depths do not share any common Web services and concepts (i.e. they are not overlap with each other). For example, dataset 1 has four sets of solutions in which the solution 1 has 10 Web services in 9 levels and the solution 2 has 21 Web

Data Set	1	2	3
Concepts	3081	3093	3081
Things	6209	6275	6222
Params	2891	45057	44720
Services	351	4131	4036
Solution Depths	9, 18, 19, 27	6, 13, 9	6
Known solutions	7023327247800	3463760	7776
Best solution	10 services/9 levels	13 services/6 levels	10 services/6 levels

Table 4: Data Sets Used in Experiments

services in 18 levels. The solution 1 and the solution 2 do not share any common Web services. In another word, the intersection of the 10 Web services in the solution 1 and the 21 Web services in the solutions 2 yields $\emptyset$. Also, none of concepts which are produced from Web services in solution 1 can contribute in finding any Web services involved in solution 2.

5.2 Planning Quality Experiments

5.2.1 Experiment Method

The purpose of this experiment is to test the performance of the planning algorithm that we developed in Section 4.6 and the quality of solutions that the algorithm produces. First, the planning algorithm generates a planning graph to tell user whether the solution is existed (Algorithm 3). Second, if the solution existed, it uses backward search process to get the solution (Algorithm 6).

5.2.2 Composition Result

We experiment our planning algorithm on the three datasets we generated in Section 5.1. We record the composition time, its time cost for both planning process and backward search process as well as their composition quality. As is shown in Table 5, "PG Composition Time (ms)" shows the time used by planning processes to construct the planning graphs; "PG Constructed" shows the number of services and

the number of levels in each constructed planning graph; "Total Composition Time" shows the total time of the planning process and corresponding backward search process for each dataset; "Extracted Solution" shows the optimized solution extracted by the backward search process for each dataset; "Best solution Existed" shows the best solution existed in each dataset. As we can see:

Data Set	1	2	3
Time to construct PG (ms)	101 (ms)	217 (ms)	154 (ms)
PG Constructed (services/levels)	125s/9l	187s/6l	57s/6l
Total Composition Time (ms)	494 (ms)	416 (ms)	4364 (ms)
Extracted Solution (services/levels)	10s/9l	13s/6l	10s/6l
Best solution Existed (services/levels)	10s/9l	13s/6l	10s/6l

Table 5: Planning Experiment Results

1. Planning process is much faster than backward search process. In dataset 3, for example, planning process only takes 3% of the total time to finish. This shows the complexity of our composition algorithm is mainly resides on the backward search process. If no solution exists, the backward search process is unnecessary.

2. The number of action levels in the planning graph generated is equal to the lowest solution depth. It is because the construction of planning graph stops when all the goals are reached.

3. Planning Graph returned by planning process contains multiple lean solutions. This is because, in our generated datasets, Web services are highly replaceable. For each concept, it might exist more than one Web service which can produce it.

4. Backward search process is able to return the optimized solution that exists in a dataset. This proves that the composition quality of our algorithm is satisfied. However, one thing we need to notice here is, although it shows in all the three results that our algorithm finds all the best solutions, it does not mean the backward search algorithm always returns the best solution. The backward search process tries to invoke only the necessary Web services at each

level by calculating the "smallest sets" of Web services to satisfy subgoals. It's possible that there are more than one "smallest set" in one level. In such case, our backward search only picks up the first one it finds and discard the rest. Therefore it exists a small chance that the final composition result it not the best one in the dataset (but is still an optimized one).

5.3 Repairing versus Re-planning Experiments

5.3.1 Experiment Method

The purpose of this experiment is to compare our repairing (Algorithm 7 and 6) with the re-planning (Algorithm 3 and 6). For re-planning, we use the same planning algorithm that we used in previous planning quality experiments (Section 5.2) but with the updated parameters. The re-planning serves as a baseline algorithm. We want to check whether repairing algorithm can be better than re-planning and, if so, under which conditions. We conduct two experiments. In Experiment 1, a certain percentage of available Web services, from 3% to 24%, are randomly removed from W. W is the set of available Web services in the Web service repository. When $n\%$ Web services are removed from W, all remaining Web services can be candidates to be added into the planning graph. In Experiment 2, a number of Web services are randomly removed from a lean solution. For example, dataset 1 has a lean solution containing 10 services, and therefore in the experiment we remove Web services from the 10 services to break the solution.

5.3.2 Experiment 1: Remove Web Services from Service Repository

The following comparison of performance is recorded in the cases that both repair (Algorithm 7 and 6) and re-planning (Algorithm 3 and 6) can find solutions. Each data point is obtained from the average of five independent runs when both repair and re-planning find a solution.

Fig 23 to Fig 26 show the results from Experiment 1. From Fig 23, we can see that the re-planning composition time is slightly decreasing when more Web services are removed. It is because the problem is smaller when less Web services are available.

However, it is more difficult to repair the plan. Therefore, the repair composition time increases in such a case. However, after a certain percentage (around 20%), the repair composition time decreases. This is because the problem becomes simpler to solve and also because we are less committed to the original composition. Please notice that repair may not find existing solutions. For example, when removing 21% Web services, we observed 4 failures on 9 runs. One thing we need to notice is, for the repairing time of re-planning in dataset 3, we see the curve decrease dramatically. This is because the dataset 3 only contains 1 solution. When we remove a number of services from that only solution, the amount of time required by backward search decrease dramatically.

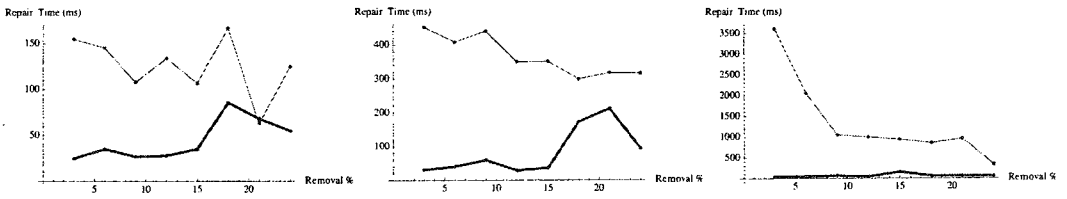

Figure 23: Repair time with dataset 1 (left) and dataset 2 (middle) and dataset 3 (right) (repair - thick line, re-planning - thin line)

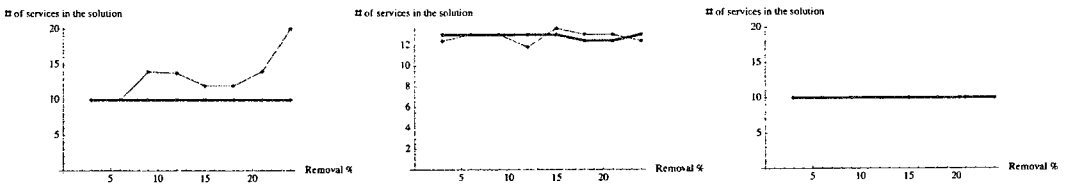

Figure 24: Number of services in the solutions with dataset 1 (left) and dataset 2 (middle) and dataset 3 (right) (repair - thick line, re-planning - thin line)

Fig 24 and Fig 25 show the number of Web services and the number of levels in a composed plan. The plot for repair is rather flat. Our explanation is that our repair algorithm does not work well when the number of levels are over 10. This is because it is a greedy search algorithm and the successful rate is lower in more level cases. As we do not count unsuccessful cases, we end up showing the cases where the levels are below 10 and very flat. Fig 24 and Fig 25 show that when repair finds a solution, the quality of the solution is pretty good.

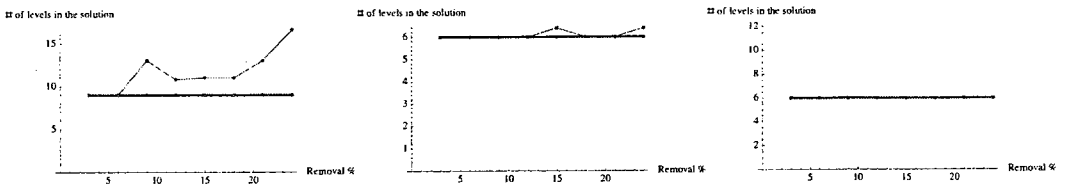

Figure 25: Number of levels in the solutions with dataset 1 (left) and dataset 2 (middle) and dataset 3 (right) (repair - thick line, re-planning - thin line)

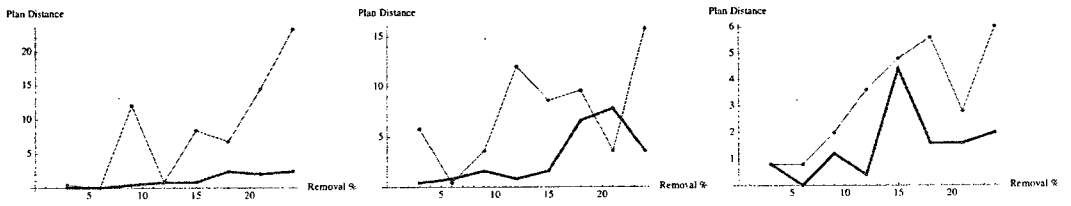

Figure 26: Plan distance to the original solution with dataset 1 (left) and dataset 2 (middle) and dataset 3 (right) (repair - thick line, re-planning - thin line)

Finally, Fig 26 shows that the solution computed with the repair algorithm can be more similar to the original plan than the solution computed with the re-planning algorithm.

5.3.3 Experiment 2: Remove Web Services from Existing Plan

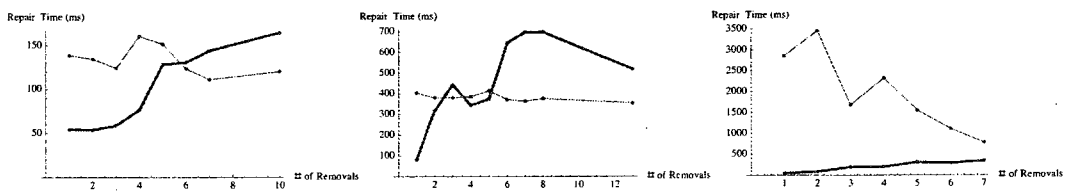

Figure 27: Repair time with dataset 1 (left) and dataset 2 (middle) and dataset 3 (right) (repair - thick line, re-planning - thin line)

Fig 27 to Fig 30 shows the results of the Experiment 2. In this experiment. Web services are removed from the solution instead of from the service repository. As is

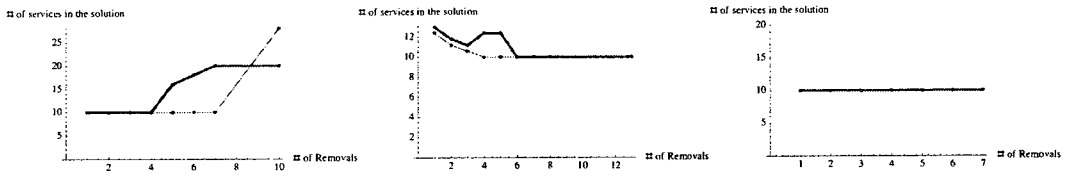

Figure 28: Number of services in the solutions with dataset 1 (left) and dataset 2 (middle) and dataset 3 (right) (repair - thick line, re-planning - thin line)

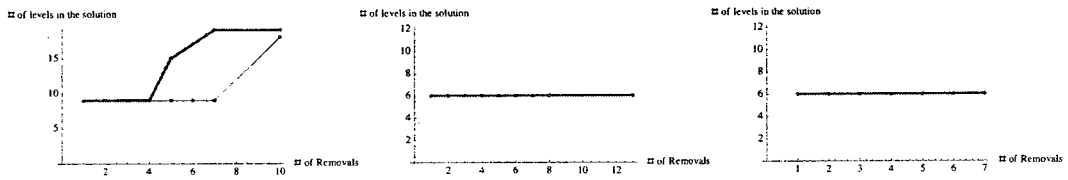

Figure 29: Number of levels the solutions with dataset 1 (left) and dataset 2 (middle) and dataset 3 (right) (repair - thick line, re-planning - thin line)

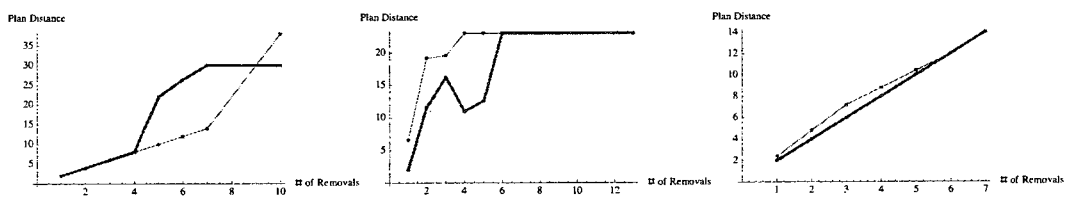

Figure 30: Plan distance to the original plan with dataset 1 (left) and dataset 2 (middle) and dataset 3 (right) (repair - thick line, re-planning - thin line)

shown in Fig 27 left and middle, the curves of re-plaining processes are relatively "stable". However, in Fig 27 right (dataset 3), we observe the curve has a significant decreasing trend. This might because dataset 3 only has one possible solution while dataset 1 and 2 have multiple solutions, when the Web services are removed from the only solution in dataset 3, the required steps of backward search decreased dramatically.

On the other hand, the curves of repairing process have clear increasing trends. This is because when the "critical" Web services are removed from the solutions, there are more BPs in the PG, and therefore become more "difficult" to repair. Also, we observe that, for all three datasets, the curves of repairing process can be lower

than the curves of re-planning when the number of removed Web services is low. the comparison of solution quality and distance to original solution is similar as in Experiment 1 and is omitted due to length limitation.

5.4 Indexed Repairing versus Re-planning Experiments

5.4.1 Experiment Method

Due to the limitation of the repairing algorithm (Algorithm 7) that we found in previous experiments (i.e. the repairing process can be trapped by some BPs that is not fixable although other solutions are exist), we made a modification by allowing indexing current solution and replacements of all Web services in the solution (Section 3.4.4). When solution is damaged, the process check the index first to see if it is fixable ("replaceable situation"). If not, it first removes all Web services that might trap the repairing process and then invoke the heuristic repairing algorithm to search a repair plan. Besides, in order to further improve the performance, in this experiment we let the repairing algorithm fix BPs so that the repairing algorithm (Algorithm 8) is able to reuse existing parts of other solutions in the "full PG".

We also redesigned our experiments for comparing our repairing (Algorithm 8 and 6) with the re-planning (Algorithm 3 and 6). The experiments changes including:

1. For each experiment run, instead of giving independent lists of randomly selected Web services to each algorithm, we now given them the same list of randomly selected Web services.

2. For each experiments, we future divide it into two categories: "replaceable situation" and "non-replaceable situation". In "replaceable situation" experiment, we only remove those Web services who have a least one replacement service. While in "non-replaceable situation" experiment, we allow remove every service existed in current solution.

3. In experiment 2, we completely remove the whole solution 1. Therefore, both repair and re-planning can go to the solution 2 which has more Web services

and levels in the solutions. The trend of curves can be more precisely shown in this case.

5.4.2 Fixing Solution 1 WSC Dataset

Experiment Method Dataset 1 has 4 solutions. Solution 1 has 9 levels and 10 Web services. Solution 2 has 18 levels and 28 Web services. Solution 3 has 19 levels and 20 Web services and Solution 4 has 27 levels and 40 Web services. In this experiment we let our algorithm fix base on one of the lean solutions we get from solution 1 by doing the planning and backward search (Algorithm 3 and 6). "non-replaceable situation" experiments randomly select a given number of Web services from all the 10 existing Web services in that lean solution. "replaceable situation" experiments only randomly select a given number of Web services from all existing Web services in that lean solution if those Web services have replacement Web services. In our case, the number of Web services in existing lean solution that have replacement Web services is 7 (out of 10). Once the list of removed Web services is selected, re-planning algorithm and our repairing algorithm (Algorithm 8) also remove the same list of Web services and then perform fixing. Notice that, for "non-replaceable situation" removal, non-indexed repairing frequently fails because of the trapped issues discussed above. Therefore, it's not in the figure

- Thin line: re-planning (Algorithm 3 and 6)
- Thick line(dark): indexed repairing (Algorithm 8 and 6)
- Thick line(light): non-indexed repairing (Algorithm 8 and 6)

As Figure 31 right ("replaceable situation") shown, when the replacements of the removed Web services exist, the indexed-repairing process works very fast due to indexing. The average composition time is about 1ms, because it only needs to check whether all removed Web services still have replacements and picks up one from them to replace the Web service. In contrast, the repairing time of non-indexed repairing processes has increasing trend and sometimes "jump up". This is because when the number of removed Web services increase the number of heuristic calculations needed also increases. Due to the heuristics, it is possible that the "non-indexed repairing" switches to different solution (solution 3 in this case) which has more levels and thus

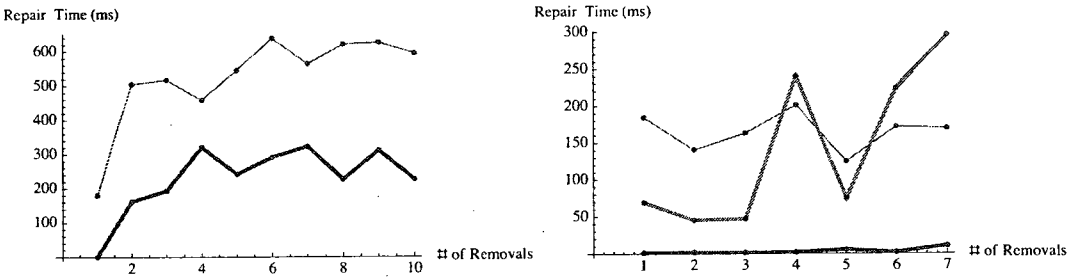

Figure 31: Repair time for fixing solution 1 "non-replaceable situation" (left), "replaceable situation" (right)

dramatically increases the repairing time. Repairing is relatively stable, the average time is 170ms which is always higher than indexed repairing process.

As Figure 31 left ("non-replaceable situation") shown, when the number of removed Web services is low, "indexed repairing" benefits from its index of replacement Web services. Therefore, the curve starts from about 1ms. When the number of removed Web services increases, repairing works but it still benefits from reusing existing parts from PG, which is faster than repairing it from sketch. Also, it is not trapped in solution 1 because it excludes the solution 1 completely when replacement Web services in the solution do not exist. For re-planning here, the time mainly depends on current Web services repository states (usually, more Web services and more levels mean more time to finish because the number of steps in backward search increases dramatically) when the number of removed Web services is low, it is very likely that the replacement Web services exist, it still return the solution 1 (which is relatively simple to solve). When the removed scope increases, re-planning returns solution 2 (which is relatively more complex) if the replacement Web service cannot be found, and therefore the curve of composition time "jumps up" at point 2 and then stays stable.

As Figure 32 and Figure 33 right ("replaceable situation") shown, when every removed Web service has replacement Web services, "indexed repairing" and re-planning only need to replace those removed Web services by using their replacement Web services. Thus, their solution has the same quality. "Non-indexed repairing" is depending on its heuristic and therefore can jumps to different solutions in some

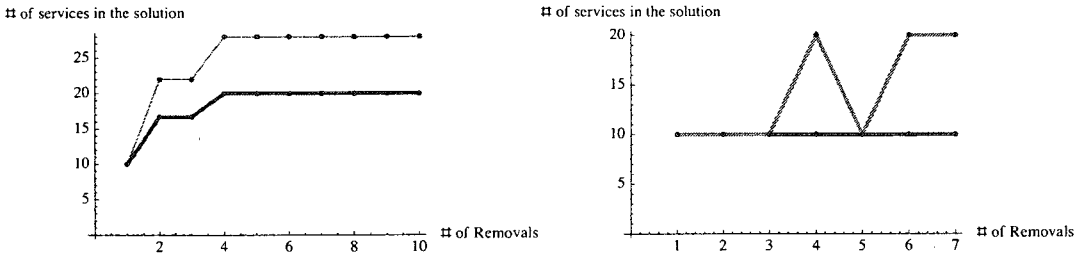

Figure 32: Numbers of services in solutions for fixing solution 1 "non-replaceable situation" (left), "replaceable situation" (right)

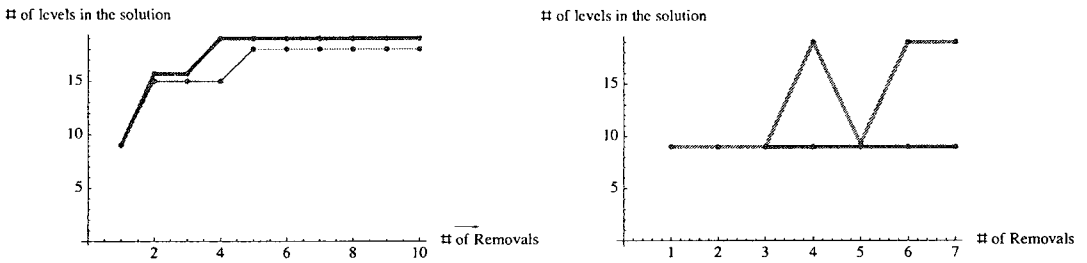

Figure 33: Number of levels in solutions for fixing solution 1 "non-replaceable situation" (left), "replaceable situation" (right)

cases. Note:

1. In Figure 32 left, the curve of re-planning is higher than the curve of repairing because re-planning goes to solution 2, which has 18 levels and 28 Web services; repairing goes to solution 3, 19 levels that has 20 Web services. Therefore, in terms of the number of Web services, the curve of re-planning is higher than the curve of repairing.

2. In Figure 33 left, the curve of re-planning is lower than the curve of repairing because re-planning goes to solution 2, which has 18 levels that has 28 Web services, repairing goes to solution 3, which has 19 levels that has 20 Web services. Therefore, In terms of the number of levels, the curve re-planning is lower than the curve of repairing.

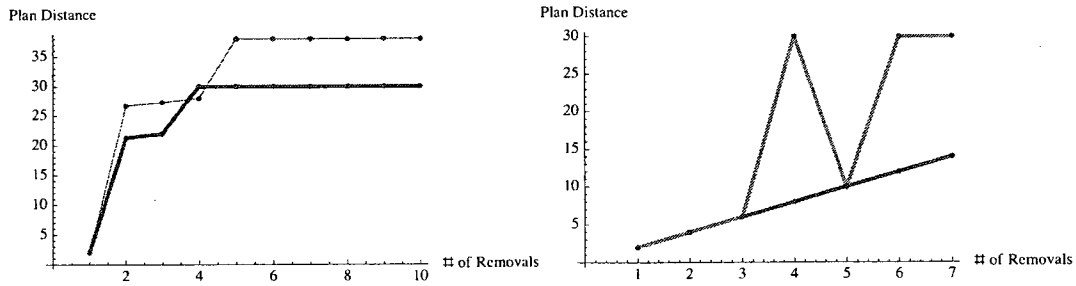

Figure 34: Plan distance to the original solution (solution 1) "non-replaceable situation" (left), "replaceable situation" (right)

For plan distance shown in Figure 34 left, re-planning and repairing are different because they goes to different solutions. Re-planning's solution has more Web services, and therefore has higher plan distance.

5.4.3 Fixing Solution 2 WSC Dataset

Experiment Method We first removed all Web services in solution 1 (both the Web services currently in solution and their replacement Web services) from Web service repository. The rest experiment method is the same as "fix solution 1 experiment" (Section 5.4.2)

- Thin line: re-planning: (Algorithm 3 and 6)
- Thick line(dark): indexed repairing (Algorithm 8 and 6)
- Thick line(light): non-indexed repairing (Algorithm 8 and 6)

As Figure 35 left ("non-replaceable situation") shown, the curve of repairing algorithm starts from about 10ms because replacement Web services can be found by looking up the index. When the number of removed Web services increases, the repairing algorithm returns solution 3 instead if the replacement Web services of solution 2 cannot be found. This is because we do not remove any Web services in solution 3, the curve is therefore flat since solution 3 is not damaged. For re-planning, the curve starts at about 600ms, and then it "switches up and downs" several times. It becomes stable after point 6. This shows when solution 2 has not been fully damaged,

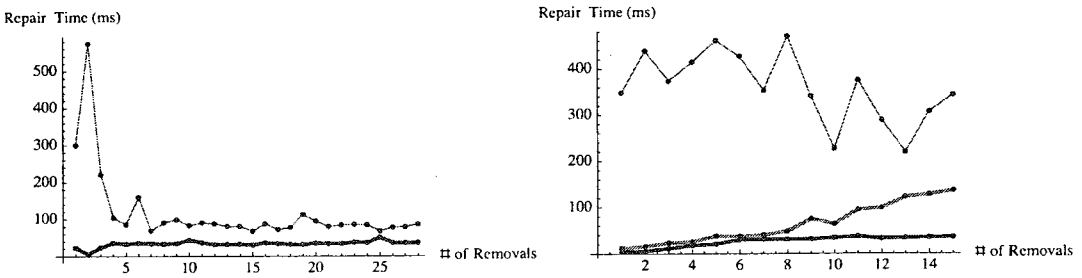

Figure 35: Repair time for fixing solution 2 "non-replaceable situation" (left), "replaceable situation" (right)

re-planning still needs to calculate and return solution 2. However, when solution 2 is damaged (the number of removed Web services is increased), it always returns solution 3. The time difference due to different steps required in backward search (due to our approach to calculate "smallest set" of Web service (Section 4.7), solution 3 is "preferred" by the re-planning than solution 2).

As Figure 35 right ("replaceable situation") shown, the "indexed repairing" is always the fastest one. The "non-indexed repairing" in this case always can find solution and it requires more time than "indexed repairing" and it has increasing trend. Re-planning always stays in solution 2 in this case therefore there is no any "switch up and down", and as the number of removed Web services increases actually it has slightly decreasing trend.

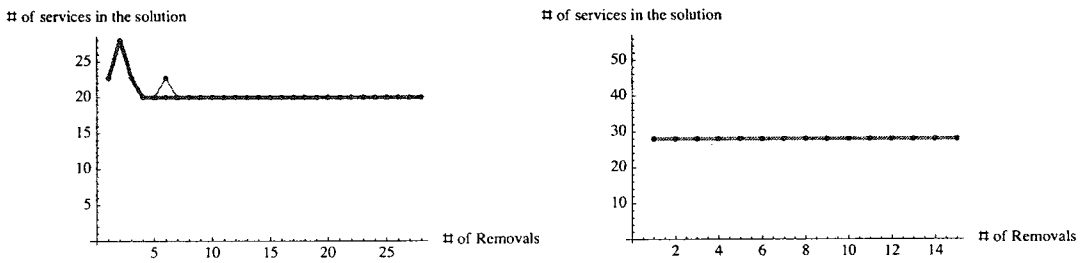

Figure 36: Numbers of services of solutions for fixing solution 2 "non-replaceable situation" (left), "replaceable situation" (right)

As Figure 36 and Figure 37 left ("non-replaceable situation") shown, re-planning and the indexed repairing almost have the same quality except at point 6, repairing algorithm returns solution 3 which contains 19 levels and 20 Web services. However, re-planning algorithm finds it not necessary to switch to different solutions, it stays in solution 2 that contains 18 levels and 28 Web services instead.

At right ("replaceable situation"), it shows re-planning and "indexed repairing" has the same quality while "non-indexed repairing" sometime switch to different solutions.

As Figure 38 right ("replaceable situation") shown, re-planning has higher distance than indexed repairing approaches, although they are based on the same solution.

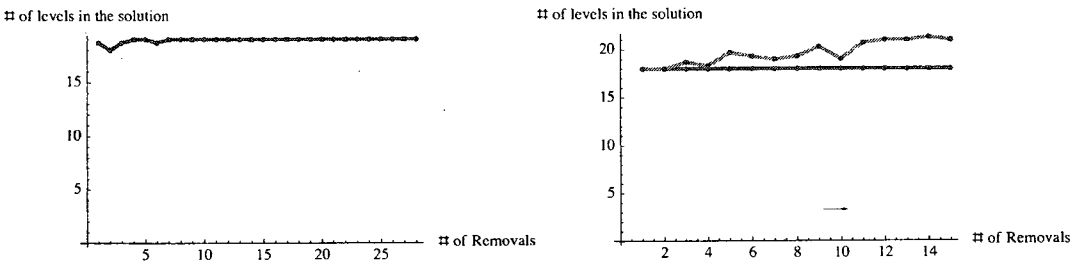

Figure 37: Number of levels for fixing solution 2 "non-replaceable situation"(left), "replaceable situation" (right)

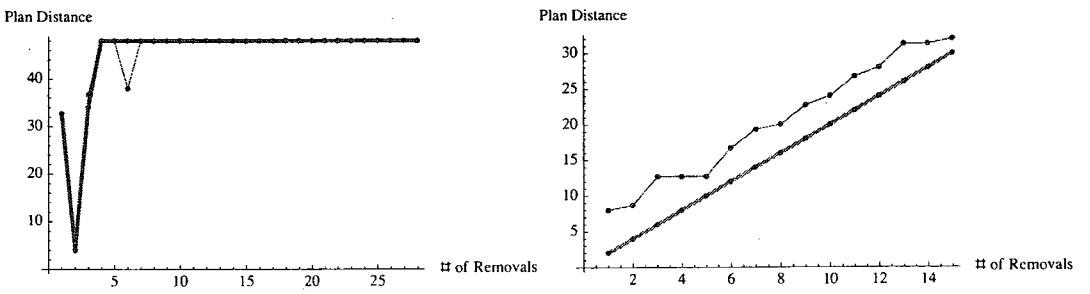

Figure 38: Plan distance to the original solution (solution 2) "non-replaceable situation" (left), "replaceable situation" (right)

scope is relatively small to the whole plan.

The results of our work show that plan repair is an attractive, as well as feasible, direction to improve the adaptation of Web service compositions in a changing world.

Chapter 6

Conclusion

This thesis begins with the introduction and brief discussion of various approaches in the Web service composition research domain, as well as their pros and cons. We carry on with one previous work [ZY08] on composition algorithms based on planning graphs, and then go one step further by putting it on a real, open-world context in which the environment changes all the time. We implement the planning graph algorithm in [GNT04] and a backward search process to extract the solution. We use them as our re-planning approach (Section 3.2) to fix the broken plan when the environment changes. We compare the re-planning approach with our repairing approaches proposed in this thesis (Section 5.3 and 5.4), which can fix the broken plan in a shorter time period while keeping the similar quality. More importantly, the plans repaired by our repair approaches have lower plan distance than the re-planning approach, which might be a critical consideration for real world businesses. Additionally, in order to further increase the success rate of our repairing approaches, we propose an index-based method which automatically removes certain broken Web services to prevent them from trapping the heuristic-based repairing processes.

For testing the performance of our algorithms, we implement all our algorithms using the popular object-oriented programming language Java. We use the platform and tools provided by Web Service Challenge 2009 [Ble10] to generate datasets that simulate the large-scale Web service composition context and validate our composition results. The experiment results show that our repairing algorithms can be better than re-planning in terms of repairing-time, quality and plan-distance when the damaged

[HBM08] Rachid Hamadi, Boualem Benatallah, and Brahim Medjahed. Self-adapting recovery nets for policy-driven exception handling in business processes. *Distributed and Parallel Databases*, 23(1):1–44, 2008.

[HM07] Seyyed Vahid Hashemian and Farhad Mavaddat. Automatic Composition of Stateless Components: A Logical Reasoning Approach. In Farhad Arbab and Marjan Sirjani, editors, *FSEN*, volume 4767 of *Lecture Notes in Computer Science*, pages 175–190. Springer, 2007.

[KPL97] Subbarao Kambhampati, Eric Parker, and Eric Lambrecht. Understanding and Extending Graphplan. In Sam Steel and Rachid Alami, editors, *ECP*, volume 1348 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 1997.

[MP09] Annapaola Marconi and Marco Pistore. Synthesis and Composition of Web Services. In Marco Bernardo, Luca Padovani, and Gianluigi Zavattaro, editors, *SFM*, volume 5569 of *Lecture Notes in Computer Science*, pages 89–157. Springer, 2009.

[MPM08] Tarek Melliti, Pascal Poizat, and Sonia Ben Mokhtar. Distributed Behavioural Adaptation for the Automatic Composition of Semantic Services. In José Luiz Fiadeiro and Paola Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2008.

[OAS10a] OASIS. Business Process Execution Language. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel, March-18th 2010.

[OAS10b] OASIS. Universal Description, Discovery and Integration (UDDI). http://www.oasis-open.org/committees/uddi-spec/faq.php, March-18th 2010.

[OLK07] Seog-Chan Oh, Dongwon Lee, and Soundar R. T. Kumara. Web Service Planner (WSPR): An Effective and Scalable Web Service Composition Algorithm. *Int. J. Web Service Res.*, 4(1):1–22, 2007.

[OOLL05] Seog-Chan Oh, Byung-Won On, Eric J. Larson, and Dongwon Lee. BF*: Web Services Discovery and Composition as Graph Search Problem. In *EEE*, pages 784–786. IEEE Computer Society, 2005.

[OWL10] OWL Pizza Example. http://www.obitko.com/tutorials/ontologies-semantic-web/owl-example-with-rdf-graph.html, March-18th 2010.

[Pee05] J. Peer. Web service composition as AI planning - a survey. technical report. Technical report, University of St.Gallen, 2005.

[SEG08] R. Seguel, R. Eshuis, and P. Grefen. An Overview on Protocol Adaptors for Service Component Integration. *BETA Working Paper Series WP 265, Eindhoven University of Technology*, 2008.

[vdKdW05] Roman van der Krogt and Mathijs de Weerdt. Plan Repair as an Extension of Planning. In Susanne Biundo, Karen L. Myers, and Kanna Rajan, editors, *ICAPS*, pages 161–170. AAAI, 2005.

[W3C10a] W3C. Hypertext Transfer Protocol (HTTP). http://www.w3.org/Protocols/, March-18th 2010.

[W3C10b] W3C. OWL web ontology language overview. http://www.w3.org/TR/owl-features/, March-18th 2010.

[W3C10c] W3C. Resource Description Framework (RDF). http://www.w3.org/TR/rdf-primer/, March-18th 2010.

[W3C10d] W3C. Semantic Annotations for WSDL and XML Schema (SAWSDL). http://www.w3.org/TR/sawsdl/, March-18th 2010.

[W3C10e] W3C. Semantic Web. http://www.w3.org/2001/sw/, March-18th 2010.

[W3C10f] W3C. Simple Object Access Protocol (SOAP). http://www.w3.org/TR/soap/, March-18th 2010.

[W3C10g] W3C. Web service. http://www.w3.org/TR/ws-arch/, March-18th 2010.

[W3C10h] W3C. Web Service Description Language (WSDL) version 2.0. http://www.w3.org/TR/wsdl20/, March-18th 2010.

[Wik10a] Wikipedia. RESTful Web Service. http://en.wikipedia.org/wiki/ Representational_State_Transfer, March-18th 2010.

[Wik10b] Wikipedia. Service Oriented Architecture (SOA). http://en.wikipedia.org/wiki/Service-oriented_architecture, March-18th 2010.

[Wik10c] Wikipedia. Simple Object Access Protocol (SOAP). http://en.wikipedia.org/wiki/SOAP, March-18th 2010.

[Wik10d] Wikipedia. Web service. http://en.wikipedia.org/wiki/Web_Service, March-18th 2010.

[Wik10e] Wikipedia. Web Services Description Language (WSDL). http://en.wikipedia.org/wiki/Web_Services_Description_Language, March-18th 2010.

[YPZ10a] Yuhong Yan, Pascal Poizat, and Ludeng Zhao. Repairing Service Compositions in a Changing World. In *SERA*, 2010.

[YPZ10b] Yuhong Yan, Pascal Poizat, and Ludeng Zhao. Self-Adaptive Service Composition through Graphplan Repair (Submitted). In *ICWS*, 2010.

[YXG08] Yixin Yan, Bin Xu, and Zhifeng Gu. Automatic Service Composition Using AND/OR Graph. In *CEC/EEE* [DBL08], pages 335–338.

[YZ08] Yuhong Yan and Xianrong Zheng. A Planning Graph Based Algorithm for Semantic Web Service Composition. In *CEC/EEE* [DBL08], pages 339–342.

[ZY08] Xianrong Zheng and Yuhong Yan. An Efficient Syntactic Web Service Composition Algorithm Based on the Planning Graph Model. In *ICWS*, pages 691–699. IEEE Computer Society, 2008.

Appendix A

Online Source Code Repository

All source code of our experiments and datasets can be download from Github at following address:
http://github.com/HappyHackingGeek/WSC09-Composition-System-Implementation

If you prefer to checkout, you will need to have Git installed in your operating system. You can use the following command to checkout a copy:
git clone git://github.com/HappyHackingGeek/WSC09-Composition-System-Implementation.git

You can find more usage about git at:
http://www.kernel.org/pub/software/scm/git/docs/user-manual.html

Appendix B

Planning Graph Algorithm

```
package ca.concordia.pga.algorithm;

import java.util.HashSet;
import java.util.Set;

import ca.concordia.pga.models.Concept;
import ca.concordia.pga.models.PlanningGraph;
import ca.concordia.pga.models.Service;

public class PGAlgorithm {

public static boolean generatePG(Set<Concept> knownConceptSet,
Set<Service> currInvokableServiceSet,
Set<Service> currNonInvokableServiceSet,
Set<Service> invokedServiceSet, PlanningGraph pg) {
int currentLevel = 0;
do {
/**
 * point knownConceptSet to pg's current PLevel
 */
knownConceptSet.addAll(pg.getPLevel(currentLevel));
currInvokableServiceSet = new HashSet<Service>();
```

```
currNonInvokableServiceSet = new HashSet<Service>();
Set<Concept> pLevel = new HashSet<Concept>();

/**
 * fetch all possible candidates
 */
for (Concept c : pg.getPLevel(currentLevel)) {
currInvokableServiceSet.addAll(c.getServicesIndex());
}
/**
 * remove those who have already been invoked
 */
currInvokableServiceSet.removeAll(invokedServiceSet);
/**
 * remove those whose invocation condition have not been satisfied
 */
for (Service s : currInvokableServiceSet) {
if (!pg.getPLevel(currentLevel).containsAll(
s.getInputConceptSet())) {
currNonInvokableServiceSet.add(s);
}
}
currInvokableServiceSet.removeAll(currNonInvokableServiceSet);
if (currInvokableServiceSet.size() <= 0) {
break;
}
/**
 * invoke the services
 */
invokedServiceSet.addAll(currInvokableServiceSet);
pg.addALevel(currInvokableServiceSet);
/**
 * generate PLevel
```

```
 */
for (Service s : currInvokableServiceSet) {
knownConceptSet.addAll(s.getOutputConceptSet());
}
pLevel.addAll(knownConceptSet);
pg.addPLevel(pLevel);
/**
 * increase the level and print out newly invoked services
 */
currentLevel++;
System.out.println("\n*********Action Level " + currentLevel
+ " *******");
for (Service s : pg.getALevel(currentLevel)) {
System.out.print(s + "|");
}
System.out.println();

} while (!knownConceptSet.containsAll(pg.getGoalSet())
& !currInvokableServiceSet.isEmpty());

return knownConceptSet.containsAll(pg.getGoalSet());
}

}
```

Appendix C

Backward Search Algorithm for Extracting Planning Graph

```
package ca.concordia.pga.algorithm;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.Vector;

import ca.concordia.pga.algorithm.utils.CombinationHelper;
import ca.concordia.pga.models.Concept;
import ca.concordia.pga.models.PlanningGraph;
import ca.concordia.pga.models.Service;

/**
 *
 * @author Ludeng Zhao(Eric)
 *
```

```
 */
public class BackwardSearchAlgorithm {


/**
 * Backward search based on PG to prune redundant web services
 *
 * @param pg
 * @return routeCouters
 */
public static Vector<Integer> refineSolution(PlanningGraph pg) {

int currLevel = pg.getALevels().size() - 1;

Set<Service> minimumServiceSet;
Set<Concept> subGoalSet = new HashSet<Concept>();
subGoalSet.addAll(pg.getGoalSet());

Set<Concept> leftGoalSet = new HashSet<Concept>();
Set<Set<Service>> routes = new HashSet<Set<Service>>();
Vector<Integer> routeCounters = new Vector<Integer>(); // debug purpose,
Map<Integer, Set<Service>> solutionMap = new HashMap<Integer, Set<Service>>();

do {
/**
 * compute services that each concept is origin from
 */
Set<Service> actionSet = pg.getALevel(currLevel);
subGoalSet.addAll(leftGoalSet);
subGoalSet.removeAll(pg.getGivenConceptSet());
leftGoalSet.clear();
for (Concept g : subGoalSet) {
for (Service s : actionSet) {
```

```
if (s.getOutputConceptSet().contains(g)) {
g.addServiceToOrigin(s);
}
}
/**
 * check if the goal can be produced by current action level
 */
if (g.getOriginServiceSet().size() == 0) {
leftGoalSet.add(g);
}
}
/**
 * defer the goals that cannot be produced in current action level
 */
subGoalSet.removeAll(leftGoalSet);

/**
 * compute all alternative routes that current level has
 */
// computeRoutes_Old(subGoalSet, routes);
computeRoutes(subGoalSet, routes);

/**
 * get the route with minimum web services to invoke
 */
minimumServiceSet = new HashSet<Service>();
Iterator<Set<Service>> itr = routes.iterator();
while (itr.hasNext()) {
Set<Service> candidate = itr.next();
if (minimumServiceSet.size() == 0) {
minimumServiceSet = candidate;
} else if (minimumServiceSet.size() > candidate.size()) {
minimumServiceSet = candidate;
```

```
}
}

/**
 * overwrite selected route into pg's ALevel (temperate
 * implementation! in order to keep pg's information)
 */
pg.setALevel(currLevel, minimumServiceSet);
solutionMap.put(currLevel, minimumServiceSet);

/**
 * get the inputs of invoked web services as subGoals
 */
subGoalSet.clear();
for (Service s : minimumServiceSet) {
subGoalSet.addAll(s.getInputConceptSet());
}
/**
 * reset routes
 */
routeCounters.add(routes.size());
routes.clear();
currLevel--;

} while (currLevel > 0);

/**
 * remove invalid concepts after pruning services
 */
int currentLevel = 1;
do{
Set<Concept> knownConceptSet = new HashSet<Concept>();
knownConceptSet.addAll(pg.getPLevel(currentLevel-1));
```

```
for(Service s : pg.getALevel(currentLevel)){
knownConceptSet.addAll(s.getOutputConceptSet());
}
pg.getPLevel(currentLevel).clear();
pg.getPLevel(currentLevel).addAll(knownConceptSet);
currentLevel++;
}while(currentLevel < pg.getALevels().size());

/**
 * debug checking if solution is valid
 */
if (leftGoalSet.size() != 0) {
System.out.println("Solution is NOT VALID!");
}

removeEmptyLevels(pg);

return routeCounters;
}

/**
 * compute removable service set from the given goal concept set
 *
 * @param conceptSet
 * @return
 */
private static Set<Service> getRemovableServiceSet(Set<Concept> conceptSet) {
...
}

/**
 * Implementation for computing all alternative routes that can produce
 * given concepts
```

```
 *
 * @param conceptSet
 * @param routes
 */
@SuppressWarnings("unchecked")
private static void computeRoutes(Set<Concept> conceptSet,
Set<Set<Service>> routes) {
...
}

/**
 * remove empty levels from given pg
 * @param pg
 */
private static void removeEmptyLevels(PlanningGraph pg){

...

}

}
```

Appendix D

Heuristic Evaluator

```
/**
 * Calculate the repairing heuristic score for one action
 * @param g
 * @param p
 * @param a
 * @return repairing heuristic score
 */
public static int evaluate(Set<Concept> g, Set<Concept> p, Service a){

int score = 0;
Set<Concept> t = new HashSet<Concept>();

/**
 * calculate g join aOut
 */
t.addAll(g);
t.retainAll(a.getOutputConceptSet());
score += t.size() * 10;

/**
 * calculate p join aIn
 */
```

```
t.clear();
t.addAll(p);
t.retainAll(a.getInputConceptSet());
score += t.size();

/**
 * calculate aIn not in p
 */
t.clear();
t.addAll(a.getInputConceptSet());
t.removeAll(p);
score -= t.size();

return score;
}
```

Appendix E

Repairing Algorithm

```
/**
 * repair given PG using backward approach
 *
 * @param pg
 * @param serviceMap
 * @param conceptMap
 * @param thingMap
 * @param paramMap
 * @return
 */
public static boolean repair(PlanningGraph pg,
Map<String, Service> serviceMap, Map<String, Concept> conceptMap,
Map<String, Thing> thingMap, Map<String, Param> paramMap) {

Set<Concept> subGoalSet = new HashSet<Concept>();
Set<Service> candidates = new HashSet<Service>();

/**
 * compute subGoalSet
 */
subGoalSet = getSubGoalSet(pg);
```

```
int currentLevel = pg.getPLevels().size() - 1;
while (currentLevel > 0 & subGoalSet.size() != 0) {

Set<Service> aLevel = pg.getALevel(currentLevel);
Set<Concept> pLevel = pg.getPLevel(currentLevel);
List<Service> sortedCandidates = new LinkedList<Service>();
Set<Concept> currentSubGoalSet = new HashSet<Concept>();

do {
/**
 * 1. compute currentSubGoalSet which contains all broken
 * preconditions and unstatisfied goals that need to be
 * statisfied right in current PLevel (level n). Initially, this
 * set will contains all unstatisfied goals only.
 */

/**
 * compute broken preconditions in current level
 */
if (currentLevel == pg.getPLevels().size() - 1) {
/**
 * initially currentSubGoalSet only contains broken goals
 */
currentSubGoalSet.clear();
currentSubGoalSet.addAll(pg.getGoalSet());
currentSubGoalSet.removeAll(pg.getPLevel(pg.getPLevels()
.size() - 1));

} else {

/**
 * compute currentSubGoalSet
```

```
 */
currentSubGoalSet.clear();
for (Service s : pg.getALevel(currentLevel + 1)) {
currentSubGoalSet.addAll(s.getInputConceptSet());
}
currentSubGoalSet.removeAll(pLevel);
}

/**
 * skip to next level if no subgoals need to be satisfied for
 * current level
 */
if (currentSubGoalSet.size() == 0) {
break;
}

/**
 * 2. Select all services that not already in current ALevel and
 * could produces at least one subgoal in currentSubGoalSet as
 * candidates. compute their heuristic score based on heuristic
 * function. Sort from highest to lowest.
 */

/**
 * compute candidate services (services not in current ALevel)
 */
candidates.clear();
for (String key : serviceMap.keySet()) {
candidates.add(serviceMap.get(key));
}
candidates.removeAll(pg.getALevel(currentLevel));
Set<Service> removableCandidates = new HashSet<Service>();
for (Service s : candidates) {
```

```
Set<Concept> outputs = new HashSet<Concept>();
outputs.addAll(s.getOutputConceptSet());
outputs.retainAll(currentSubGoalSet);
if (outputs.size() == 0) {
removableCandidates.add(s);
}
}
candidates.removeAll(removableCandidates);

/**
 * compute heuristic score for each of candidate service
 */
sortedCandidates.clear();
for (Service s : candidates) {
int score = RepairingEvaluator.evaluate(currentSubGoalSet,
pg.getPLevel(currentLevel - 1), s);
s.setScore(score);
sortedCandidates.add(s);
}
Collections.sort(sortedCandidates, serviceScoreComparator);

/**
 * 3. Insert first service in candidate list to current ALevel.
 * Add its outputs to PLevel(n). Check if PLevel(n) still has
 * unstatisfied subgoals. if yes, repeat step 1 unless
 * candidates list become empty then return unrepairable.
 */
if (sortedCandidates.size() == 0) {
return false;
}
Service candidate = sortedCandidates.get(0);
sortedCandidates.remove(0);
aLevel.add(candidate);
```

```
pLevel.addAll(candidate.getOutputConceptSet());
// pg.getPLevel(currentLevel-1).addAll(candidate.getInputConceptSet());
System.out.println("CurrentSubGoalSet size: "
+ currentSubGoalSet.size());

currentSubGoalSet.removeAll(pg.getPLevel(currentLevel));

System.out.println("Added: " + candidate + "("
+ candidate.getScore() + ")" + " at: " + currentLevel);
System.out.println("sortedCandidates size: "
+ sortedCandidates.size());
// System.out.println("CurrentSubGoalSet size: " +
// currentSubGoalSet.size());

} while (currentSubGoalSet.size() != 0
& sortedCandidates.size() > 0);

/**
 * if currentSubGoalSet cannot be satisfied, return unrepairable
 */
if (currentSubGoalSet.size() != 0) {
return false;
}

/**
 * 4. compute subGoalSet which contains all broken preconditions and
 * unsatisfied goals based on current PG status, if empty return PG
 */

/**
 * compute subGoalSet based on current PG status
 */
subGoalSet = getSubGoalSet(pg);
```

```
if (subGoalSet.size() == 0) {
return true;
}

System.out.println("subGoalSet size: " + subGoalSet.size());
/**
 * 5. decrease level count by 1
 */
currentLevel--;

/**
 * 6. if level == 0, insert a new PLevel which contains only the
 * given concepts to Level 0 (all current Plevel number increased by
 * 1), increse level count by 1.
 */
if (currentLevel == 0) {
pg.insertPLevel(0, new HashSet<Concept>());
pg.getPLevel(0).addAll(pg.getGivenConceptSet());
pg.insertALevel(0, new HashSet<Service>());
currentLevel++;
}

}

/**
 * Upon success: Start from level 1 while level < levels.size() 1.
 * select each service in current ALevel, removed all its duplicate in
 * higher ALevel. 2. increse level count by 1
 */

currentLevel = 1;
while (currentLevel < pg.getALevels().size()) {
for (Service s : pg.getALevel(currentLevel)) {
```

```
int higherLevel = currentLevel + 1;
while (higherLevel < pg.getALevels().size()) {
Set<Service> duplicates = new HashSet<Service>();
for (Service hs : pg.getALevel(higherLevel)) {
if (s.equals(hs)) {
duplicates.add(hs);
}
}
if (pg.getALevel(higherLevel).removeAll(duplicates)) {
System.out.println("duplicates removed!");
}
higherLevel++;
}
}
currentLevel++;
}

return true;
}
```

Appendix F

Sample Execution Log of Composition Process using Planning Graph Algorithms

```
Initializing Time 2851
Concepts size 3081
Things size 6209
Param size 2891
Services size 351

Given Concepts:
con649167057 | con1272418610 | con440870358 | con2056039516 | con1990498516 |
con2034534667 | con605130906 | con1131301455 | con1451017854 | con369185398 |
con1629000785 | con1494027552 | con736419041 | con1531919743 | con731911495 |
con1518811543 | con1754553008 | con1064121911 | con97600426 | con1636783251 |
con1935404941 | con243633175 | con108659942 | con1146866387 | con1498535060 |

Goal Concepts:
con510302591 |

*********Action Level 1 *******
```

serv783934155|serv296269980|serv365702213|serv1056747493|serv926075671|
serv856643438|serv1058386037|serv1818863550|serv853366388|serv233391847|
serv91250331|serv1751069823|serv1546050174|serv302824080|serv21818098|
serv364063707|serv995507904|serv1688191690|serv714501922|serv163959614|
serv1618759457|serv988953804|

*********Action Level 2 *******
serv1757623923|serv1064940137|serv1820502056|serv1476617941|serv1888295783|
serv1126179726|serv433495940|

*********Action Level 3 *******
serv1265044192|serv1896488389|serv1195611959|serv441688546|serv435134446|
serv572360406|serv1127818270|serv372256313|serv641792639|serv1334476425|
serv2027160249|serv502928173|serv160682564|serv1957728016|serv1134372370|
serv1827056156|serv1615482407|

*********Action Level 4 *******
serv511120779|serv922798621|serv2096592482|serv1203804603|serv2035352855|
serv1273236836|serv649985245|serv580553012|serv1197250503|serv1889934289|
serv230114797|serv1342669069|serv1684914640|serv1965920622|serv992230854|

*********Action Level 5 *******
serv26733692|serv299547030|serv18541048|serv1266682736|serv96165925|
serv504566679|serv711224872|serv1412101302|serv2104785088|serv719417478|
serv788849711|serv1959366522|serv1481533535|serv1403908658|serv1473340891|

*********Action Level 6 *******
serv368979263|serv1131095320|serv507843729|serv1550965768|serv1893211339|
serv1200527553|serv438411496|serv2028798755|serv1823779106|serv1336114969|
serv573998912|serv1061663087|serv1542773124|serv87973281|serv858281944|
serv780657105|serv1754346873|serv165598158|

*********Action Level 7 *******

serv1269959786|serv2098230988|serv643431145|serv1962643572|serv712863378|
serv2032075805|serv1620398001|serv1405547202|serv850089338|serv577275962|

*********Action Level 8 *******
serv235030391|serv1339392019|serv927714177|serv304462624|serv1612205357|
serv20179592|serv1474979435|serv1689830234|serv997146410|serv157405514|

*********Action Level 9 *******
serv1408824252|serv1681637590|serv1759262467|serv646708195|serv919521571|
serv782295611|serv1544411668|serv2101508038|serv1066578643|serv226837747|
serv89611825|

=========Goal Found=========
PG Composition Time: 101ms
Execution Length: 9
Services Invoked: 125
============================

==================================
===========After Pruning==========
==================================

*********Action Level 1 (alternative routes:3) *******
serv1056747493

*********Action Level 2 (alternative routes:3) *******
serv1126179726

*********Action Level 3 (alternative routes:7) *******
serv1195611959
serv502928173

*********Action Level 4 (alternative routes:1) *******

```
serv2096592482

*********Action Level 5 (alternative routes:4) *******
serv18541048

*********Action Level 6 (alternative routes:3) *******
serv87973281

*********Action Level 7 (alternative routes:1) *******
serv850089338

*********Action Level 8 (alternative routes:2) *******
serv1612205357

*********Action Level 9 (alternative routes:3) *******
serv919521571

=================Status=================
Total(including PG) Composition Time: 494ms
Execution Length: 9
Services Invoked: 10
==================End==================
```