AUTOMATIC WEB SERVICE COMPOSITION WITH AI PLANNING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MEHMET KUZU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JULY 2009

Approval of the thesis:

AUTOMATIC WEB SERVICE COMPOSITION WITH AI PLANNING

submitted by **MEHMET KUZU** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Müslim Bozyiğit _____
Head of Department, **Computer Engineering**

Assoc. Prof. Dr. Nihan Kesim Çiçekli
Supervisor, **Computer Engineering Dept., METU** _____

Examining Committee Members:

Assoc. Prof. Dr. Ali Doğru
Computer Engineering Dept., METU _____

Assoc. Prof. Dr. Nihan Kesim Çiçekli
Computer Engineering Dept., METU _____

Assoc. Prof. Dr. Halit Oğuztüzün
Computer Engineering Dept., METU _____

Asst. Prof. Dr. Pınar Şenkul
Computer Engineering Dept., METU _____

Dr. Gökçe Banu Laleci Ertürkmen
SRDC, METU _____

Date: 27.07.2009

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Mehmet Kuzu

Signature :

ABSTRACT

AUTOMATIC WEB SERVICE COMPOSITION WITH AI PLANNING

Kuzu, Mehmet

M.S., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Nihan Kesim Çiçekli

July 2009, 129 pages

In this thesis, some novel ideas are presented for solving automated web service composition problem. Some possible real world problems such as partial observability of environment, nondeterministic effects of web services, service execution failures are solved through some mechanisms. In addition to automated web service composition, automated web service invocation task is handled in this thesis by using reflection mechanism. The proposed approach is based on AI planning. Web service composition problem is translated to AI planning problem and a novel AI planner namely "Simplanner" that is designed for working in highly dynamic environments under time constraints is adapted to the proposed system. World altering service calls are done by conforming to the WS-Coordination and WS-Business Activity web service transaction specifications in order to physically repair failure situations and prevent undesired side effects of aborted web service composition efforts.

Keywords: Automatic Web Service Composition, Automatic Web Service Invocation, Semantic Web Services, AI Planning

ÖZ

YAPAY ZEKA PLANLAMA TEKNİKLERİ İLE OTOMATİK WEB SERVİS BİLEŞİMİ

Kuzu, Mehmet
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Nihan Kesim Çiçekli

Temmuz 2009, 129 sayfa

Bu tezde, otomatik örün servis birleşimi problemini çözmek için bazı yenilikçi düşünceler sunulmuştur. Çevrenin kısmen algılanabilirliği, örün servislerinin belirsiz etkileri, servislerin uygulama anı hataları gibi bazı gerçek hayat problemleri birtakım yöntemler aracılığı ile çözülmüştür. Otomatik örün servis birleşimine ek olarak, otomatik örün servis yürütme işlemi de dinamik programlama özellikleri kullanılarak bu tez kapsamında ele alınmıştır. Önerilen yaklaşım tarzı, yapay zekâ planlama tekniğine dayalıdır. Örün servisleri birleşimi problemi, yapay zekâ planlama problemine dönüştürülmüş; son derece dinamik ortamlarda ve zaman kısıtlaması altında çalışmak için tasarlanmış yenilikçi bir yapay zekâ planlama aracı olan "Simplanner" sisteme uyarlanmıştır. Çevresel değişim etkileri olan servis çağrıları, WS-Coordination ve WS-Business Activity örün servis hareketleri tanımlamalarına uygun olarak gerçekleştirilmektedir. Böylece hatalı durumların fiziksel onarımı yapılıp, yarıda kesilen örün servis birleşim denemelerinin istenmeyen yan etkileri ortadan kaldırılmıştır.

Anahtar Kelimeler: Otomatik Örün Servis Birleşimi, Otomatik Örün Servis Tetiklenmesi, Anlamsal Örün Servisleri, Yapay Zekâ ile Planlama

To my family

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude and appreciation to my supervisor Assoc. Prof. Dr. Nihan Kesim Çiçekli for her endless encouragement and support throughout this study. I am very lucky that I have such a friendly, intellectual, patient and benignant supervisor.

I would like to thank Assoc. Prof. Dr.Nilufer Önder who gave me invaluable suggestions about my thesis. I want also thank to Çağla Okutan and Ertay Kaya who work on the same subject and helped me a lot for overcoming difficulties.

I am deeply grateful to my parents who devoted their life to their children and my brother Ali Cem for their love and support. Without them, this work could not have been completed.

I am deeply indebted to my friends, Gökhan Yaprakkaya, Hüseyin Yılmaz, Kurtçebe Eroğlu, Çağlar Şenaras, Özgür Karaaslan, Nedim Ozan Tekin and all the other colleagues at HAVELSAN whose suggestions and encouragement helped me a lot during my study.

I would thank the Scientific and Technological Research Council of Turkey (TÜBİTAK) for providing the financial means throughout this study.

Finally, my special thanks go to Asst. Prof. Dr. Oscar Sapena who is the creator of Simplanner that is the most important component of this thesis. He answered all my questions with patience and clarifies all the obscure points in my mind.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

Distributed computing is widely used in today's software systems as it provides more scalable, more fault tolerant environment and it enables ready to use software components for others. In distributed software environment, interoperability is very important. The resources that are provided on the web are generally used by important amount of software applications. Loose coupling between those resources and consumer software applications should be assured, otherwise environment becomes unmanageable. Web services are the most important tools of distributed computing as they provide required loose coupling between distinct systems and interoperable distributed software environment. Web services provide ready to use functionalities through fixed interfaces for other applications by hiding the implementation details and they are commonly used in real world applications.

Web is growing very fast and important amount of web services are available to use. Although a huge number of web services exist, sometimes they are not sufficient to satisfy user needs. Inexistence of a web service that responds to the user request does not mean that request cannot be handled. Web services can collaborate to satisfy the user request, that is, some new functionality can be produced by composing the existing functionalities to handle user requests. Since the number of web services and the possible collaboration scenarios are huge, manual analysis on them for achieving the user goal is very difficult and beyond the human ability. This web service composition process should be done by software agents automatically. The main goal of this thesis is to design and implement a software agent that automates the web service composition task in an effective manner in terms of time and adaptability to the real world environment.

The aim of the proposed software agent is to find several web services and execute them according to discovered execution order for handling the user request, given a repository of services and the user goal. There exist two commonly accepted approaches for solving the web service composition problem: workflow composition and AI planning [1]. AI planning approach is more flexible and adaptable, so that approach is widely accepted in the community. AI planning approach is adapted for the construction of the web service composer agent in this thesis.

The motivation of this thesis is to solve some of the open issues of web service composition (WSC) problem. WSC problem involves distinct subproblems. First, the web service composition problem should be mapped to an AI planning problem. In order to do this mapping pure syntactic definition is not sufficient, semantic annotation is needed as well. Generally web services are described using WSDL [2] which presents the required physical execution information such as service endpoint, network communication protocol and syntactic service input and output message definitions. Those definitions are used by the web service invoker component of the proposed software agent but not the AI planner component. AI planner needs semantic information for solving the stated problem and that semantic information is provided by OWL/OWL-S ([3], [4]) semantic web languages. The whole aim of semantic web is to make web machine interpretable and ontology web language (OWL) is the most important and commonly accepted tool for describing web resources semantically. In this thesis, the software agent tries to fulfill user requests, which are stated in OWL, by using the web services that are marked with OWL-S semantic service descriptions. AI planners mostly work with PDDL [5] data format like the SIMPLANNER [6] that is used in this thesis. Mapping the service composition problem to the AI planning problem requires some translation between distinct data formats, which is OWL-S/OWL to PDDL mapping in this work.

After web service composition problem is mapped to the AI planning problem, AI planner should reason on the planning problem and produce a valid plan for satisfying the user needs. AI planner produces plans in an abstract level that is; it only considers semantic objects and their relations but not the syntactic counterparts.

As a third step, abstract level solutions should be made concrete by using the syntactic information that is obtained from other resources such as WSDL definitions, users and other web services. Concrete plans should then be executed to handle the user request. The proposed software agent makes use of reflection mechanism to invoke services automatically. During execution, some unexpected situations may arise such as service unavailability because of network failures or information unavailability. The software agent should handle such undesired situations. The proposed software agent is capable of handling such unexpected cases with the help of the used AI planner, namely SIMPLANNER and adapted WS-Business Activity framework [7].

Automated web service composition problem is a hot research topic. There are some excellent surveys ([8], [9], [10]) about the challenges of web service composition (WSC). Although many important works have been conducted in order to solve this challenging problem so far, still there exist many open issues. In [10], some open issues are presented about the available WSC solutions that have been proposed up to year 2004. Many of those stated issues are still open. Some of the stated problems in [10] and the solutions to those problems provided as contributions of this thesis can be briefly described as follows:

- If unexpected events that cause failures occur, replanning and some compensation mechanisms are required.

  In this thesis, this problem is solved in two levels. In the first level which is an abstract logical level, a novel AI planner that is specially designed for highly dynamic, nondeterministic environments is used. The proposed solution interleaves planning and execution. Planner keeps track of the current state by using the initially provided semantic state description and the semantic effects of executed actions. That state description includes all information about the environment. If something unexpected occurs, the web service composer agent informs the planner about the situation such as service unavailability through the provided semantic statements. The planner will then change the current state according to that information and does dynamic replanning for handling the

unexpected situation. If the planner is able to find a new solution after replanning, according to the new situation, the composition task continues. Sometimes it is possible that solution for handling the unexpected situation does not exist. In such a case the planner informs the service composer agent about the inexistence of a solution for the new situation. If planner cannot find a new solution for the new state, the second level handling mechanism takes place.

The second level is physical level and failure recovery is provided by using the WS-Business Activity framework. After the planner provides some abstract actions which are some semantic action definitions that are grounded with semantic objects, the service composer agent constructs physical counterparts with reflection mechanism by using some information resources such as WSDL and users themselves. When web services become ready to be physically invoked, some more analysis is conducted by the software agent. If the service to be invoked has some world altering affects, its call is not directly done but handled in collaboration with WS-Business Activity [7] coordinator as specified in WS-Business Activity specification. If some failure occurs which cannot be handled by the AI planner, software agent informs the coordinator which then sends required compensation signals to the participant web services. As a result, aborted service composition sessions do not cause undesired side effects.

- Environment that is available during planning may not be the same as the environment during execution which is problematic. Interleaving planning and execution is desirable.

In this thesis, the proposed web service composer agent interleaves planning and execution. Planning problems are generally very difficult problems to solve and generally requires exponential computation time. Huge time requirement is not acceptable since users look for responsive systems. For timely response, planners generally use some admissible heuristics and some decomposition techniques. Especially for decomposition, huge amount of domain knowledge is required which is not available most of the times. Another technique that is used for timely response is any time planning approach. In any time planning

approach, the planner constructs a very quick initial solution which may include some wrong decisions, and improve that initial solution if time is available [6]. The adapted AI planner for this thesis namely “SIMPLANNER” is any time planner. It concentrates not on the total plan but the best initial action for the solution. The service composer agent requests that best initial action from the planner and does the real execution operation after doing the other required steps. During real service execution, planner continues to plan for finding a new best action. Generally service execution time is sufficient for the planner to produce the new best action. As a result interleaving planning and execution is done in a timely manner by using the features of Simplanner.

- Web services may have some nondeterministic effects which should be tackled.

The used AI planner allows state changes when required at any time during composition. If services have some nondeterministic effects, they can be observed by the service composer agent after real execution and state change request is made to the planner. The planner changes the state according to the occurred effect and continues to plan by considering that effect.

- Service descriptions are provided once and they are not updated frequently. From time to time some availability checks should be done.

The proposed solution interleaves planning and execution as mentioned before. Initially it is assumed that, all service descriptions are valid. Some logical statements that describe service availability are presented in the initial logical state, and preconditions of action definitions are updated with a logical statement that shows the availability of that particular action. After a real service call, if there exists some problem about the service. The service composer agent informs the planner and makes the logical counterpart of the problematic service unavailable. As a result, the planner does not consider that problematic service for further decisions and try to solve the problem using alternative services.

- It is possible that, web services can produce new objects at run time. It is very difficult to model dynamic object generation with current AI planners, some other mechanism should be provided for handling such cases.

Logical actions can only be proposed by the planner to execute if the preconditions of those actions are satisfied and required logical objects for grounding the logical action are available. Sometimes, it is possible that such logical objects are not available in the initial state but can be produced by other services. For handling this case, the service composer agent constructs a logical object for each defined type and adds a logical statement to the initial sate in order to assert that users can provide the details of logical objects at run time. The precondition of each action is updated with the requirement that logical action parameters should be known before execution. If initial assumption does not hold, that is, the user cannot provide the details of logical objects, current logical state is updated. If the value of the service parameter cannot be provided by the user, the precondition of that particular action cannot be satisfied. In such a case, planner tries to find some other service that supplies those objects and changes the plan accordingly. Existence of one logical object for each type is not a problem, because in this level an abstract plan is constructed. Before the real service call, the real values are obtained from users or other web services and those values are destroyed after usage. One logical object in the planner level may correspond to multiple objects in the execution level. Some memory data structures are constructed for handling dynamic object generation.

- Semantic web service descriptions should be connected to the real web services.

Semantic web service descriptions are high level definitions that are used by the AI planner in this thesis. For real web service interaction, syntactic counterparts of these semantic service descriptions are needed. The bridge between the syntactic and semantic definitions is provided by using the definitions stated in the grounding section of OWL-S descriptions. The service composer agent that is proposed in this thesis makes use of that section and WSDL definitions of

services for real service interaction. Real values of the syntactic counterparts are obtained either from the user or from other web services. By substituting the real values to the syntactic interfaces through reflection mechanism, service composer agent does the real interaction with web services.

In 2007, IBM Research has published an article about the approaches for web service composition and execution [11]. In [11], some case studies presented along with the desirable properties of web service composition solutions. These desirable features are required in most of the real world scenarios. In the following, these features are summarized and the ability of the proposed system to satisfy these desirable features is discussed.

- Adaptability: Changes in run environment should be handled.

This thesis proposes highly adaptable web service composition solution. Interleaving planning and execution, and modifying the plan according to the outcomes of the executed services and user interactions continuously, make the proposed solution highly flexible.

- Failure Resolution: Web service composition is done in a very dynamic and non reliable environment. As a result some failures may occur from time to time. Such undesired situations should be handled.

One of the main goals of this thesis is to discover and resolve failures. The software agent is able to detect failures by making real service calls and by interacting with the user. If a failure situation is detected, recovery mechanisms are fired both in logical level and physical level. That is; logical state definitions are modified according to the encountered failure and if required, physical compensation is done through WS-Transaction frameworks ([7], [12], [13]).

- User Interaction: The user should be capable of supervising the software agent.

This thesis gives importance to user interaction. Initially users provide their requests in an abstract way and give details by interacting with the software

agent continuously. According to the abstract level goal definition, software agent finds some high level solutions and the details of high level solutions are asked to the user during run time. Users have the option to say that they do not know the required information and direct the software agent to find some services that provide the particular information. After service execution, the results can be observable by the user. Although not implemented yet, a mechanism can be provided that enable users to invalidate some services manually according to their results easily by using the same approaches as automatic invalidations.

- Scalability: There exist a huge amount of relevant and irrelevant services. This huge search space should be examined in a short period of time.

Scalability is a very challenging issue, especially for the domain independent AI planners. Domain independent AI planners cannot respond if the search space is too big. Since domain knowledge is very limited for WSC, domain dependent planners cannot be adapted as well. Most of the current solutions like this one work on some particular domain not on the whole web. Scalability issue is not tackled in this thesis but considered as a future work. Some filtering mechanisms should be applied to the available services according to the user request and irrelevant services should be eliminated to some extent. The software agent that is proposed in this work assumes that those filtering are done a priori.

This thesis tries to solve some open issues about automated web service composition and invocation problems as mentioned above. The organization of the thesis is as follows: In Chapter 2 some background information about the used technologies is presented. Some important works that have been conducted for the solution of WSC problem are described and their strengths and weakness with respect to this work are discussed. In Chapter 3, the general system architecture is presented. In Chapter 4, adapted AI planner, namely “Simplanner” is introduced and its application to WSC domain is described. In Chapter 5, automated web service invocation issues and proposed action caching mechanism are presented. In Chapter 6, transactional issues are discussed, WS-Transaction specifications are introduced and their integration to

the system architecture is presented. In Chapter 7, a case study is conducted that clarifies the functionality of the proposed system. Finally in Chapter 8, the thesis is concluded and some future works are discussed.

CHAPTER 2

BACKGROUND INFORMATION AND RELATED WORK

This chapter contains two parts. In the first part, some background information about the general concepts and terminology that is used in this thesis work is presented. In addition, the most important methodologies and the general system architecture that is applied for solving the web service composition problem are described in this part.

In the second part, basic ideas behind the relevant work in the literature about web service composition with AI planning approach are described. Their strengths and weaknesses are discussed with respect to this thesis work.

2.1 Background Information

In this part, first some web service related concepts are described. Section 2.1.1 gives information about web services and section 2.1.2 describes OWL-S which is the semantic web service description language. Then, concepts related to AI planning are discussed. In section 2.1.3, AI planning is presented and in section 2.1.4, PDDL data format which is the input language of many AI planners is described. Lastly, in section 2.1.5 web service composition problem is defined and some general WSC arhitectures are presented.

2.1.1 Web Services

Interoperability between distributed and distinct systems is very important in today's software applications. Service oriented architectures and the core of SOA "web services" are the commonly accepted approaches for building interoperable applications.

The formal W3C definition of the web services is stated as follows “A Web service is a software system designed to support interoperable machine-to-machine interaction over a network” [14]. Web services are provided by some service providers and consumed by software agents according to some predefined rules. Web service providers present a functional interface for service consumers without giving any information about the internal implementation details of the service which assures loose coupling between different applications. Service clients can make use of provided services for their business needs in any hardware and operating system environment and with any programming language.

Web services provide interoperability by means of assuring to conform to some standards. Web service specification declares standards such as WSDL for service descriptions, SOAP for messaging format and UDDI registry for presenting services to the community.

WSDL:

Web Service Description Language (WSDL) [2] is a language to define a web service syntactically. It allows defining input and output requirements of the service and it provides the physical location of the service (the host address and port combination). The formal WSC definition of the WSDL is stated as follows “WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information” [2]. According to WSDL specification, it contains the following elements [2]:

- **Types:** A container for data type definitions using some type system.
- **Message:** An abstract, typed definition of the data being communicated.
- **Operation:** An abstract description of an action supported by the service.
- **Port Type:** An abstract set of operations supported by one or more endpoints.

- **Binding:** A concrete protocol and data format specification for a particular port type.
- **Port:** A single endpoint defined as a combination of a binding and a network address.
- **Service:** A collection of related endpoints.

SOAP:

The message exchange format between a web service and its client is determined by Simple Object Access Protocol (SOAP) [15] specification. The formal W3C definition of SOAP and it is components is stated as follows "SOAP is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses" [15]. SOAP is not a network communication protocol such as http, but an application level protocol that is used in conjunction with the available network communication protocols.

SOAP message is transformed in XML format between collaborating parties. The exchanged XML messages contains three parts that are soap envelope, soap header and soap body. Envelope is the wrapper for the entire message, body is the container for the main information that is exchanged between peers and header contains some auxiliary, system-wide information such as transactional and security attributes.

UDDI:

Universal Description Discovery and Integration (UDDI) [16] registry is a kind of yellow pages of newspapers. It presents the available service definitions to the community in a central location. Therefore it becomes possible for business partners to find each other through provided service interfaces. The UDDI specification is determined by OASIS, for which formal definition is as follows: "Universal Description, Discovery and Integration, or UDDI, is the name of a group of web-

based registries that expose information about a business or other entity and its technical interfaces (or API's). These registries are run by multiple Operator Sites, and can be used by anyone who wants to make information available about one or more businesses or entities, as well as anyone that wants to find that information." [16].

2.1.2 OWL-S

Current web technology is generally based on syntactic constructs which prevents machine interpretability of the web. This necessitates human intervention for many of the web based tasks such as B2B and B2C applications. Web grows very fast. Every day thousands of new services are provided in web environment which makes manual operations on it so difficult, so making web, machine understandable is very important. For that purpose, semantic web and its mostly accepted language Ontology Web Lamguage (OWL) [3] has been constructed. The current syntactic web will disappear and it will be replaced by semantic web in near feature. From web services point of view, semantic web enables automatic service discovery, automatic service composition and automatic service invocation. There exist some semantic service description languages such as WSMO [17], SESMA [18] and OWL-S [4].

OWL-S which is based on OWL is the widely accepted semantic web service description language in the literature. OWL-S specifies a higher ontology for semantic web service descriptions. This ontology consists of three parts Service Profile, Service Model and Service Grounding [4].

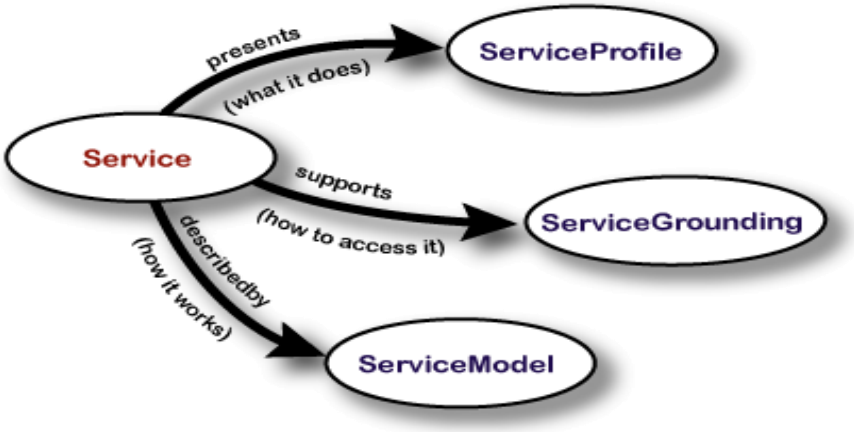

Figure 2-1 OWL-S Model

Figure 2-1 that is adapted from [4] shows the visual representation of the OWL-S components. According to OWL-S specification, the use of these components is as follows:

ServiceProfile: "The service profile tells "what the service does", in a way that is suitable for a service-seeking agent to determine whether the service meets its needs. This form of representation includes a description of what is accomplished by the service, limitations on service applicability and quality of service, and requirements that the service requester must satisfy to use the service successfully" [4].

ServiceModel*: "*The service model tells a client how to use the service, by detailing the semantic content of requests, the conditions under which particular outcomes will occur, and, where necessary, the step by step processes leading to those outcomes. That is, it describes how to ask for the service and what happens when the service is carried out" [4].

ServiceGrounding: "Service grounding specifies the details of how an agent can access a service. Typically grounding will specify a communication protocol, message formats, and other service-specific details such as port numbers used in contacting the service. In addition, the grounding must specify, for each semantic

type of input or output specified in the ServiceModel, an unambiguous way of exchanging data elements of that type with the service" [4].

2.1.3 AI Planning

According to [19], the task of coming up with a sequence of actions that will achieve a goal is called planning. Search based and propositional or first order logic based software agents can be thought as AI planning agents, but they are very primitive in the sense that they cannot be used in big domains, thus in real world applications. For instance, the instantiation of a simple service that provides direction between two addresses is infeasible for a particular problem if the number of available addresses is high, since the search space is the square of the number of available addresses. If there exist 1000 addresses, the search space contains 1 million nodes. So some heuristics and some useful domain knowledge are needed for extracting such heuristics.

AI planners use some sort of languages such as PDDL [5] that give information about the domain and the problem itself. By using the representational power of these languages, important heuristics are extracted that prune the search space. A planning problem is generally described by state and action combinations. States are conjunctions of some positive and negative literals that describe the world, and actions are the operators that can change the available state to another state. Actions have preconditions and effects. In order to execute an available action, its preconditions should be satisfied in the current state and if an action is executed, the current state changes with the effects of the executed action.

There exist different planning paradigms. The basic ones are based on state space search such as forward chaining and backward chaining [19]. In forward chaining, a search is conducted for reaching the goal state from initial state and in contrary to this approach, backward chaining starts the search from goal state and tries to reach the initial state. Generally speaking, backward chaining is better than the forward chaining since it has lower branching factor. Both approaches find total order plans, and do not use effective heuristics, so they cannot be used in most of the real world problems because of their high computational complexity. A better approach is used

by partial order planners (POP). Instead of finding total plans and considering the problem as a whole, POP subdivides the problem into smaller parts which decreases the computational complexity significantly [19]. A better one is graphplan, which makes use of very important heuristics such as mutex relations between literals and actions that are extracted from planning graphs, so it guides its search more intelligently [20]. The presented approaches are the basic underlying planning algorithms of various planners that are used in real world domains. Based on the ideas of presented general planning techniques, real world planners considers time constraints, nondeterminism, partial observability and scalability issues and make important additions to the basic algorithms. For instance HTN planners [19] are very similar to partial order planners but they make use of task decomposition which provides scalability in real world by reducing the time complexity, but they need some additional domain information to achieve this.

2.1.4 PDDL

PDDL [5] is the de-facto standard that is used as an input language by most of the AI planners [19]. It has sublanguages for STRIPS [21], ADL [22] and HTN domains. In this work STRIPS sublanguage is used. PDDL definitions are provided in two parts: domain.pddl and problem.pddl. In domain pddl, available actions and predicates are defined and in problem.pddl initial state, goal state and available objects are defined. As stated in [23], the format of domain.pddl is as in Figure 2-2.

```
(define (domain <domain name>)
        <PDDL code for predicates>
        <PDDL code for first action>
        [...]
        <PDDL code for last action>
)
```

Figure 2-2 PDDL Domain File Format

Predicates represent the object and data type properties that exist between objects. Actions represent the semantic meaning of the operations and they give information about the precondition and effects of the operations as well as input and output specifications. The format of the problem.pddl is presented in Figure 2-3.

```
(define (problem <problem name>)
    (:domain <domain name>)
    <PDDL code for objects>
     <PDDL code for initial state>
     <PDDL code for goal specification>
)
```

Figure 2-3 PDDL Problem File Format

Objects represent the available physical and conceptual components. Initial state represents the current states of the available objects. The goal representation shows the desired state of the available objects.

2.1.5 Web Service Composition

There exist many web services that are provided for some particular needs. These services can be combined to achieve more complex tasks what is called web service composition (WSC). The aim of service composition is to find several web services for obtaining an unavailable but a desired service. WSC is a hot research topic and there exist considerable amount of work in this area. The ones that are most similar to this thesis work are presented in section 2.2. The methodologies and general system architecture for web service composition that are adapted from many surveys about the topic (i.e. [24], [1], [11]) is summarized in this part.

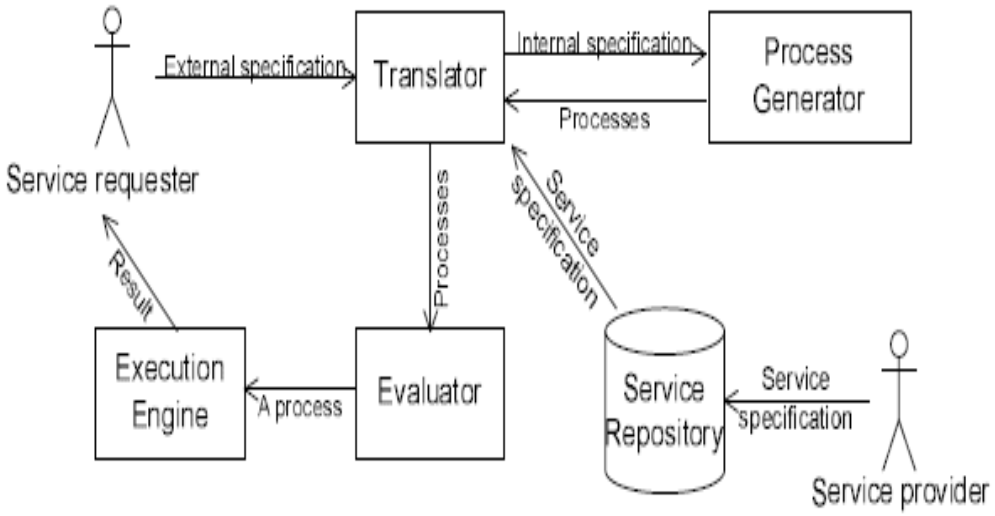

Figure 2-4 Web Service Compostion Framework

Figure 2-4 that is adapted from [1] shows the general architecture of WSC frameworks. Most of the works that have been conducted on this topic (as well as this thesis) conforms to the presented abstract framework. The presented general framework has five phases that are as follows [1]:

Presentation of single service*:* In this phase service signature is presented by service providers. The input and output parameters of the service as well as its semantic descriptions such as preconditions, information providing and world altering affects are provided. In addition to functional properties, nonfunctional attributes are described such as transactional and quality of service attributes. In this thesis WSDL is used for syntactic definitions and OWL-S is used for semantic descriptions. As a nonfunctional attribute transactional information is provided in WSDL files. Generally UDDI registries contain mentioned information of services in a central repository, but in this thesis local file system is used as a service repository.

Translation of the languages*:* Generally, the language that is used for semantic web service descriptions and problem description languages of the clients are not directly interpretable by the solution providing engine: workflow engines and AI planners. Therefore usually a translation is needed for solution providing engines to understand the problem and available domain knowledge. In this thesis, service description language is OWL-S, initial state and goal description language is OWL and the language that is used by the used planner is PDDL. So a transformation between OWL-S/OWL to PDDL is required.

Generation of composition process model*:* This phase is the most important part of the WSC where workflow engines and AI planners come into play. The approaches applied in this part are the most important one that forms the distinction between different works. In this phase a solution for the problem of users is found. In this work a novel AI planner namely Simplanner is used for composite process generation.

Evaluation of composite service*:* In some cases, there exist more than one solution to the presented problems. In such cases human intervention can be needed or evaluation engine can decide on the better plan according to some heuristics. In this work Simplanner decides which to choose; human assistance is not used.

Execution of composite service: After a total solution or some partial solution is found to the problem, actions can be executed in the physical environment. After actions are determined, client stubs execute actions through rpc calls to the services. In this work, planning and execution phases are interleaved so the execution begins without total plan generation.

There exist two commonly accepted approaches for WSC that are using workflow engines and that are using AI techniques [1]. Although workflows engines have been used in some works for WSC such as in EFlow [25], they are not as successful as the AI methodologies for WSC because important amount of human intervention is needed for workflow engines and their flexibility is limited. For instance in EFlow [25] case, the abstract service composition is generated manually; the only automation that is provided is to bind those abstract service definitions to the

concrete services. Some more dynamic workflows engines exist that are applicable to WSC but they too require an important amount of human intervention as in Polymorphic Process Model (PPM) [26] case. In this work some abstract service composition is provided apriori but in addition to it, state transitions that are caused by the operations are stated as well which makes some flexibility possible by means of reasoning on the state machine.

Most of the approaches for automated WSC are using AI planning techniques. The general models of AI planners defined and the applicability of their instances to WSC domain are discussed according to some important evaluation criteria [24].

From web service composition point of view, the core requirements of AI planners are determined as follows in [24]:

- **The domain complexity should support a significant subset of ADL:** The planner namely Simplanner used in our work is based on STRIPS [21] language but it supports some ADL [22] constructs such as type assignments to variables. Almost all ADL sentences can be converted to STRIPS statements.

- **Support for complex goals i.e. hints that tell the planner which actions should precede which other actions:** This is a challenging task that is not supported by Simplanner. Many of the available planners cannot handle complex goals that require iteration, selection or some ordering constraints. The ones that support complex goals to some extent generally need important amount of human intervention.

- **The ability to deal with incomplete information:** One of the most important characteristics of the Simplanner is its ability to operate on partially observable domains.

- **Related to the problem of supporting sensing actions is the ability of planners to dynamically add (or remove) objects to (or from) the**

domains*:* Simplanner enables to add and remove literals, as well as changing the values of numeric values any time.

- **There is a strong need for dealing with the nondeterministic behavior of services: Web service operations may fail during execution time or they may yield unexpected or undesired results:** Simplanner keeps track of the states and it is able to change to previous state if something goes wrong. It can also do replanning to tolerate the unexpected situations. Simplanner is one of the most suitable planners that can be used in a nondeterministic environment as in WSC case.

Figure 2-5 and Figure 2-6 that are adapted from [24] show the characteristics of commonly used AI planners with respect to the mentioned evaluation criteria that proves the importance of Simplanner in WSC domain.

Planner	Domain complexity	Extended Goals	Sensing	Dynamic Objects	Nondeterm. actions
FF (state based)	PDDL 2.1 level 1	No	No	No	No
FF-Metric (state based)	PDDL 2.1 level 1, level 2	No	No	No	No
HSP 2.0 (state based)	PDDL/ADL without complex precond./goals	No	No	No	No
IPP (Graphplan based)	PDDL/ADL	No	No	No	No
SGP (Graphplan based)	PDDL/ADL, without complex negations in precond./goals	No	Support sensing actions that determine the truth value of formulas	No	Produces contingent plans (?)
STAN4 (Graphplan, state based)	The STRIPS+Equality subset of PDDL	No	No	No	No
VHPOP (POP based)	PDDL 2.1 level 1 and 3	No	No	No	No
BLACKBOX (SAT, Graphplan based)	PDDL/STRIPS with restrictions	No	No	No	No
LGP (SAT based)	PDDL 2.1 levels 1,2,3	No	No	No	No

Figure 2-5 WSC Requirements vs. Neoclassical Planners

Planner	Domain complexity	Extended Goals	Sensing	Dynamic Objects	Nondeterm. actions
SHOP2 (HTN based)	PDDL/ADL with metrics and time	yes, as HTN Methods	HTN methods may contain explicit sensing actions	?	HTN Methods can be designed to deal with nondet. actions
ConGolog (High level prog.exec.)	Sit.Calc.	yes, as Golog program executions, incl. userdef. constraints	Golog program may contain sensing actions/subgoals	?	Golog programs can be designed to deal with nondet. actions
MIPS (Planning as Mod.Check.)	PDDL/STRIPS + negative preconditions and univ. conditional effects	supports CTL	No	No	No
MBP (Planning as Mod.Check.)	PDDL2.1+extensions	temporally extended goals as supported by NuPDDL	Yes	no ?	Yes, can create strong(cyclic) or weak plans
TLPlan (Temporal)	ADL + metrics	temporally extended goals as supported by MITL	No	No	No
TALPlanner (Temporal)	PDDL 2.1 (or TAL)	TAL narratives	Yes	?	No (under development?)

Figure 2-6 WSC requirements vs. Advanced Planners

Planner	**Domain Complexity**	**Extended Goals**	**Sensing**	**Dymamic Objects**	**Nondeterm. Actions**
Simplanner	PDDL 2.1	No	Yes	?	Yes

Figure 2-7 WSC requirements vs. Simplanner

In Figure 2-7, Dynamic Objects is represnted with “?”. In addition to the features of planner, application level support is needed for dynamic object representation.

2.2 Previous Works with Similar Approaches

There exist important previous works related to automated web service composition with AI planning. In this part, the most successful ones are briefly described and compared with this work.

2.2.1 Web Service Composition with SHOP2 HTN Planner

Hierarchical Task Network (HTN) planning approach is adapted for web service composition domain in [27]. SHOP2 [28] HTN planner is used for finding plans for achieving goals of the user. In this work DAML-S semantic service descriptions is used for constructing SHOP2 axioms, which are then given input to the planner along with the problem and goals to be achieved. Generally, since HTN planning is fast, it is preferred more than any other planning algorithm with respect to time requirements.This makes it very suitable for real world problems which contain huge search space as in the WSC case. Although HTN planners have very important performance advantages, they need reasonable amount of domain knowledge which is a strict requirement. Some DAML-S descriptions include domain knowledge such as information about the subtasks of a particular action in the process model part of it, but this is not the case most of the time. Using the additional information that is presented in the process model is a good idea for making planning process more effective but constructing the whole idea on top of the availability of such information is not so reasonable. The main information source of DAML-S descriptions that should be used for service discovery or composition purposes is profile part of DAML-S and that part contains atomic service descriptions only which does not provide task hierarchy that gives the main value to HTN based planners. In [27], execution and planning is interleaved as in our work, but with some limitations both in DAML-S service descriptions and in procedure. They only execute information providing services during plan generation in order not to alter the world state during plan generation. In order to achieve this, they have some strict

assumptions about service descriptions. They make it compulsory to have only one of world altering effects or information providing effects in DAML-S descriptions. Executing world altering affects is not acceptable before total plan generation, since it may cause unintentional results if the whole plan could not be achieved. In our work we have solved this problem by using WS-Coordination [12] and WS-Business Activity [7] frameworks which guarantee the atomicity of composed services. In our system there is no need to have any assumptions for DAML-S descriptions, and our work is able to totally interleave planning and execution. However, our work does not make use of the sub-process definitions that may be available in process model part of the OWL-S description.

2.2.2 Web Service Composition with OWLS-XPLAN

In [29], a WSC model is proposed based on Xplan and OWLS2PDDL conversion. OWLS2PDDL mapping is something trivial and it enables the usage of many planners for WSC which is very important. In SHOP2 [27] case, OWLS descriptions are converted to SHOP2 axioms, so the problem can only be solved using SHOP2 planner. In this case a much more general problem statement approach is used by using PDDL. Most of the planning algorithms can be used after the first phase (owls2pddl mapping) instead of Xplan. As mentioned before, if hierarchically structured action execution patterns is available, HTN planners outperforms the other planners but if this is not the case, the action based planners are better. XPlan is constructed based on this idea. It is a hybrid planner that is built on action based Fast Forward planner [30] with HTN component. OWLS-XPLAN finds the whole plan before execution so it does not work well on partially observable domains and it is not time efficient solution. Apart from these problems, this work does not handle nondeterminism that is naturally available in web services. If a found plan is interrupted because of some reason such as a network failure, a new plan is needed to be generated from scratch. More seriously, if some actions with world altering affects are executed before the interruption of the plan, inconvenient and undesired states may be constructed. This thesis has common characteristics with [29] as the initial phase of the WSC process which is mapping between owls and pddl. For that

part, we have used the owsl2pddl mapping convention and ideas that are presented in [29] but with some extensions that are necessary in nondeterministic environments. In contrast to [29], this thesis both interleaves planning and execution. In addition, the transactional properties are preserved in this thesis which prevents undesired states. This thesis proposes more flexible and adaptable solution for real world environments with respect to [29]. However, [29] is more scalable than our solution since Xplan contains HTN components.

2.2.3 Web Service Composition with WSPLAN

In work [31] and [32], a different approach is proposed. They also use PDDL as input language to the planner but they propose a totally new idea for semantic web service annotation. Instead of using OWL-S, they constructed a new semantic description language, called SESMA [18] that is designed for WSC purposes. It has the same power as OWL-S with regards to representational aspects but it is not based on OWL. The main advantage of using SESMA is its simplicity but this is not worth to use it, since there exist a de-facto standard that is accepted by the semantic web community which is OWL-S. After mapping SESMA to pddl, they have used AI planning too, but with a different approach.

According to [31], using one planning approach for all WSC tasks is not appropriate. For instance, if resource optimization is required Metric-FF [30] planner is more suitable; if planning with typed variables and lifted actions is required VHPOP [33] is more suitable; and if durative actions are required LPG [34] planner is better [31]. Because of these facts, they propose to plug in any planner instead of using a particular planner. Although the idea seems good initially, it is not concrete. Planners will have different characteristics in fact but in order to use the differences between them, one needs to understand which one to use for a particular problem by inspecting the properties of the problem. This is a very challenging task for machines and it is not examined in [31]. The work talks about nondeterminism in WSC but the proposed solution is a naïve one. Handling nondeterministic cases is ascribed to application logic which is totally contradictory to the service oriented architecture. They have also mentioned about complex goal definitions that include

selection and iteration constructs. However, the proposed solution is not a desirable one; they propose to embed these constructs into the application logic with hard coding. This thesis has many advantages with respect to [31] in terms of adaptability to dynamic and unreliable environments. Complex goal definitions are allowed in [31] which is not the case in this thesis. However, the proposed complex goal definition mechanism of [31] is not a desirable one because of the previously mentioned problems.

2.2.4 IBM's Service Creation Environment Based on End to End Composition of Web Services

The approach in [35] is different from the other existing work and this thesis. The main focus is not to satisfy user needs as in our case but to construct new services by using the existing ones by means of BPEL [36] constructs. In order to construct a new service, first its functional (input, output, precondition, effect) and nonfunctional requirements (quality of service, time constraints, etc) are described. Then a two-phase process is applied for service composition: logical composition and physical composition. Logical composition provides functional requirements of the new service that are described through OWL-S. The execution engine does not exist in this work, so problems that can arise during execution are not handled. Logical composition is done by using Planner4J framework [37] which is a kind of contingency planner that adds some branches to the plan according to distinct outcomes of sensing actions. After logical composition, physical composition phase starts, in which concrete service matching is done. In logical composition some sort of type matching is conducted. There may exist more than one concrete service compositions that provide such a type matching. The choice between multiple possible services is done in physical composition phase according to nonfunctional attributes that are stated in new service request. The most attractive part of this work is the filtering part. They propose to eliminate irrelevant services before planning, Available AI planners cannot deal with too many actions which makes WSC applicable only in a particular domain but not in the whole web. They mention about an important experiment. On a problem having a 7-step plan, the planner can return a solution in 4 seconds if filter is enabled when 100,000 irrelevant service

types/actions are present. However it takes hours without a filter. Filtering determines the irrelevant actions with the specified goal, but the underlying algorithm about this filter and its computational complexity are not presented.

CHAPTER 3

OVERALL SYSTEM ARCHITECTURE

In this chapter, an overall architecture of the proposed automated web service composition and the service invocation infrastructure are presented along with a motivating example. The main component of the proposed system is Simplanner [6] which is an effective domain independent AI planner. Simplanner works with PDDL [5] data format as most of the other on the shelf planners. Therefore, before using Simplanner, semantic service descriptions and user requests are converted from OWL-S format to PDDL data format. The translation of OWL/OWL-S format to PDDL format is described in detail in this chapter.

3.1 Motivating Example

Today web is growing very fastly. A variety of new services are added to the web everyday. The problem is that, it is very difficult for human beings to find out the relevant services and their collaboration requirements for achieving their goals. The whole aim of the semantic web is to make web machine interpretable in order to make such difficult problems solvable by machines. As a result, humans do not need to conduct manual analysis on the web for achieving their goals which is a very time consuming task. Instead they only provide their requests to the software agents which handle the problem automatically.

Manual analysis on web services for discovering the relevant ones and their possible collaborations for achieving a request is very difficult because of the existence of a huge search space. In addition to this difficulty, some more problems exist such as unreliability and partial observability of the web.

Software agents should also consider these difficulties during handling the user request, otherwise unintentional side effects might be generated.

A sample scenario that clarifies the mentioned problems can be stated as follows: Suppose a user is planning to travel; he/she needs to reserve a flight from a source airport to a destination airport in a specific time period and he/she also needs to reserve a vehicle transport from his/her home to the source airport. Both of these two requirements are atomic. If requested flight is not provided, vehicle transport should not be booked either. If the user cannot reserve vehicle transportation to the airport, there is no way for him/her to reach airport so flight booking should not be done in such a case either. In this scenario, there is a need to use six different web services in order to achieve the user's goal. The required services and their functions are listed in Figure 3-1. The user provides his/her request logically; he/she says that "I want to book flight from A to B and I want to book transportation from C to A". The details of the request are determined at run time by the software agent. For this problem a particular solution that will be provided by the software agent is shown in Figure 3-2.

CreateVehicleTransportAccount: This service creates a vehicle transport account for a particular person. The user provides personal information such as name, address, password, etc. and the service creates a transport account that is required for reservation in later stages.

RegisterPersonWithTransport: This service reserves a particular transport to a particular person. It requires that person has a transport account and it requires transport id information. If requirements of the service are provided, it books the transport for the person.

RequestTransport: This service provides a transport id corresponding to request parameters such as source and destination locations and arrival time.

CreateFlightAccount: This service creates a flight account for a particular person. The user provides personal information such as name, address, password, etc. and the service creates a flight account that is required for reservation in later stages.

BookFlight: This service reserves a particular flight to a particular person. It requires that person has a flight account and it requires flight id information. If requirements of the service are provided, it books the flight for the person.

ProposeFlight: This service provides a flight id corresponding to request parameters such as source and destination locations and arrival time.

Figure 3-1 Example Services in Motivating Example

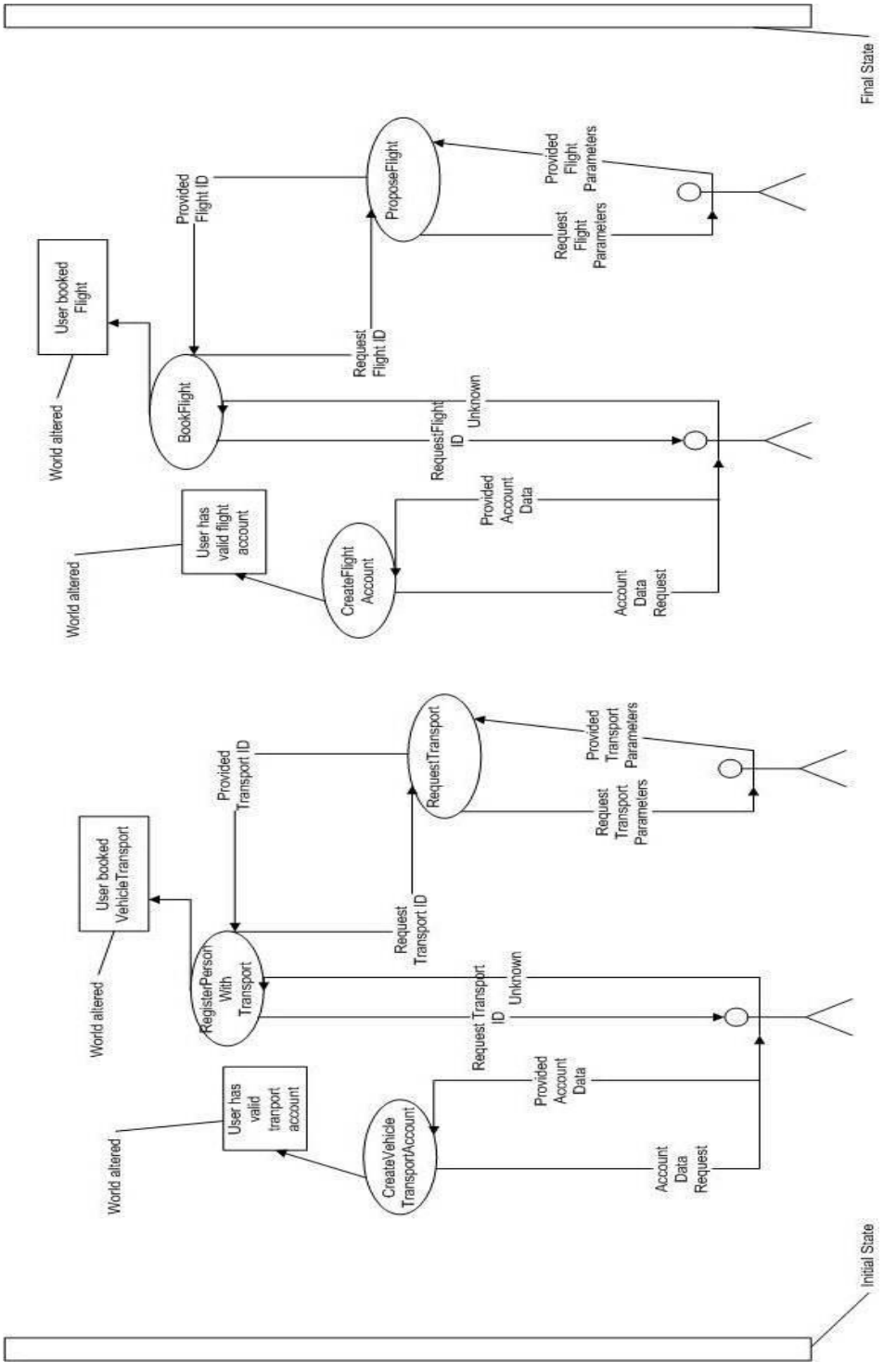

Figure 3-2 Motivating Example

“BookFlight” and “RegisterPersonWithTransport” are the main services that solve the user’s problem. These two services require “validtransport account” and “validflight account” that are provided by the “CreateFlightAccount” and “CreateVehicleTransportAccount” services. Also booking services need information that is “flight id” and “transport id” respectively. If this information cannot be provided by the user which is the case in this scenario, “ProposeFlight” and “RequestTransport” services provide that information according to user provided flight and transport parameters.

Initially, “CreateVehicleTransportAccount” service is fired. It prompts the user account data request which is the physical counters (obtained from WSDL) of logical account data that is provided in domain ontology. The user provides the required information and the service executes. As a result of the execution, a valid transport account is created for the user. Then, “RegisterPersonWithTransport” service is fired and requests transportid information from the user. Since the user could not provide the requested transportid information, another service, namely “RequestTransport” service is discovered by the software agent which provides the needed information. The user provides required transport parameters such as source and destination locations, and the service produces a transportid. This information is conveyed to the “RegisterPersonWithTransport” service and the service is fired again with its requirements satisfied. At the end, a vehicle transport is reserved for the user. Similar operations are conducted for the other goal which is booking a flight.

During all these operations some unexpected situations may arise such as service execution failure or non observable information. For instance in this scenario, during booking operation, transport and flight ids are needed which are not known by the user. Luckily, another service can provide this information so a valid solution can be extracted. If such a service could not be discovered, session should be terminated. This problem occurs for service failures as well, if service execution fails, it is not considered as a valid service and another alternative service is tried to be found for achieving the user’s goal. If such a service could not be found, the session should be aborted again. Such unintentional failures are very dangerous; they may cause side

effects in the state of the world, if world altering services are executed up to failure point in the session. In this particular scenario, four of six services produce world altering affects. For instance, in this scenario assume that there is a problem with "CreateFlightAccount" service. Up to the execution of that service, three services are executed. A vehicle transportation account is constructed for the user and a particular transport is booked for him/her. During "CreateFlightAccount" execution, if a problem occurs and another solution could not be found by the software agent, travel bookings should be postponed to another time. The problem is that two side effects are generated by the previously executed services which should be rolled back. In our system, world altering executions are done in a transactional environment and roll back operations are conducted to prevent undesired side effects of unsuccessful solution attempts.

This sample scenario shows the motivation of the system that is presented in this thesis. Even in such a simple problem, six different services are used and some collaboration is conducted between them. Extracting these services and most importantly discovering the collaboration between them for achieving a goal is cumbersome for human beings in a short period of time. Software agents should be used for performing such difficult tasks. However, software agents should consider the unexpected situations; otherwise they cannot be used in real world problems since undesired results cannot be accepted in most of the real world problems.

The proposed system in this work can solve the mentioned scenario automatically without any human assistance. The user only states her goal and provides the required information for web service arguments. The system can also handle the possible unexpected situations effectively.

3.2 System Architecture

In this subsection, the general system architecture of the proposed automated web service composition and invocation framework is presented. The architecture that is described in this section is a high level architecture. The details of the system will be explained later. The general architecture is composed of five phases which are

preprocessing, planning, action handling, executing and unexpected event handling phases. The proposed architecture provides a highly dynamic, interactive, flexible and time efficient service composition and invocation framework.

3.2.1 Preprocessing Phase

Preprocessing phase prepares all the required objects for the upcoming phases. In this phase, the system uses semantic and syntactic service descriptions that are located in the service repository, user provided domain ontology, and the initial state and goal state ontologies. These information resources are used for producing the required information in specific data format and the required service client codes. The graphical representation of this phase is given in Figure 3-3.

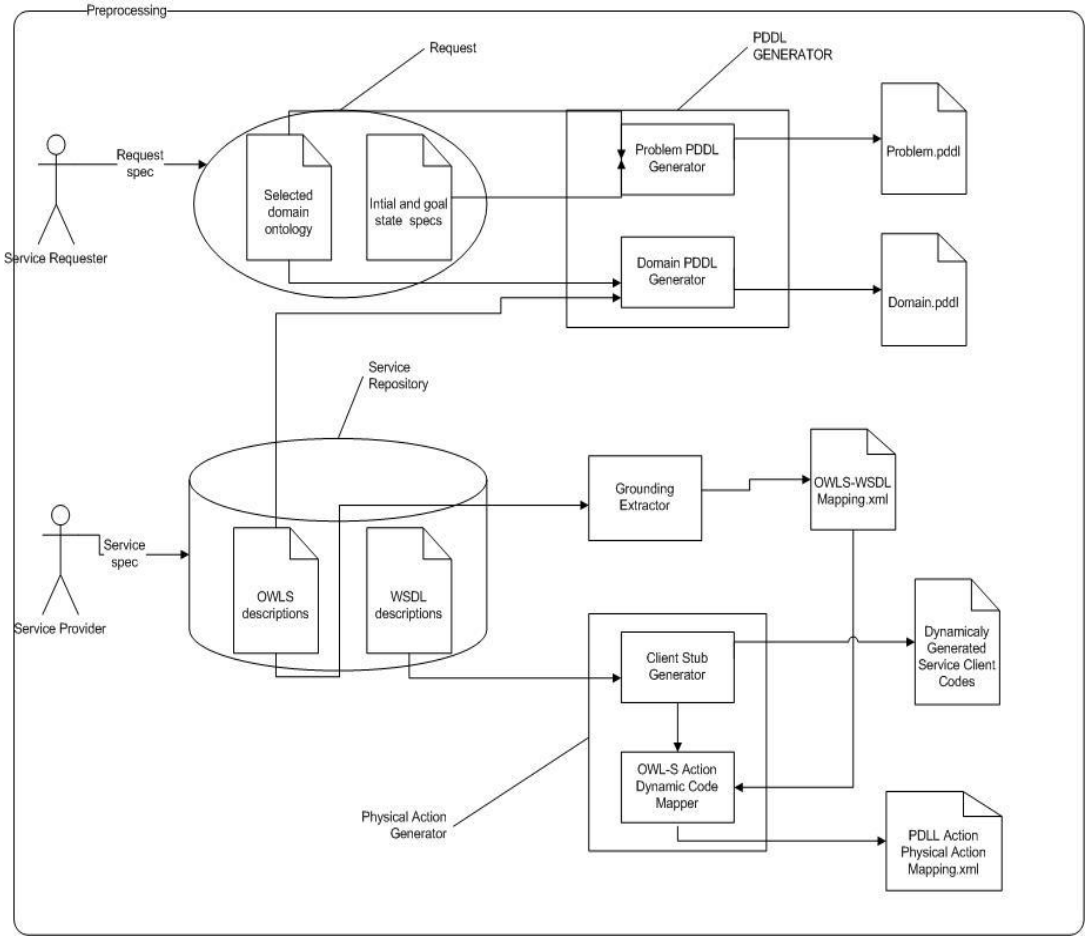

Figure 3-3 Preprocessing Phase

In this phase, the software agent produces domain knowledgebase in PDDL data format by using the OWL-S semantic service descriptions that are provided by the service providers in the service repository and selected domain ontologies by the service requester. The software agent also produces problem information in PDDL data format by using the selected domain ontology and the user request. The user request is provided by using OWL, the initial state and goal state are represented by the help of OWL. The user request can contain both world altering and information gathering requests. Generated information in PDDL specification will then be used by the AI planner in planning phase.

Another important operation of this phase is the generation of physical counterparts of logical actions that are represented in domain knowledgebase. In order to achieve this, initially OWL-S-WSDL mapping is extracted from the grounding part of the OWL-S files in order to find syntactic requirements of domain actions and their parameters. In the physical state, logical actions and their parameters are represented by machine interpretable codes and types respectively. By using WSDL descriptions of services, client stubs are automatically generated which contain both service client implementations and complex type implementations that are defined in WSDL files by the service providers. As a last step PDDL action- Physical action mapping information is constructed by using the automatically generated codes and OWLS-WSDL mapping. This information will then be used by the executor in the service execution phase.

3.2.2 Planning Phase

In this phase, all required work is handled by Simplanner. Initially, Simplanner does grounding by using the available PDDL objects and PDDL action definitions and produces the possible logical action instances. After grounding, it constructs an initial plan and continues to plan in the lifetime of the session. The graphical representation of the planning phase is described in Figure 3-4.

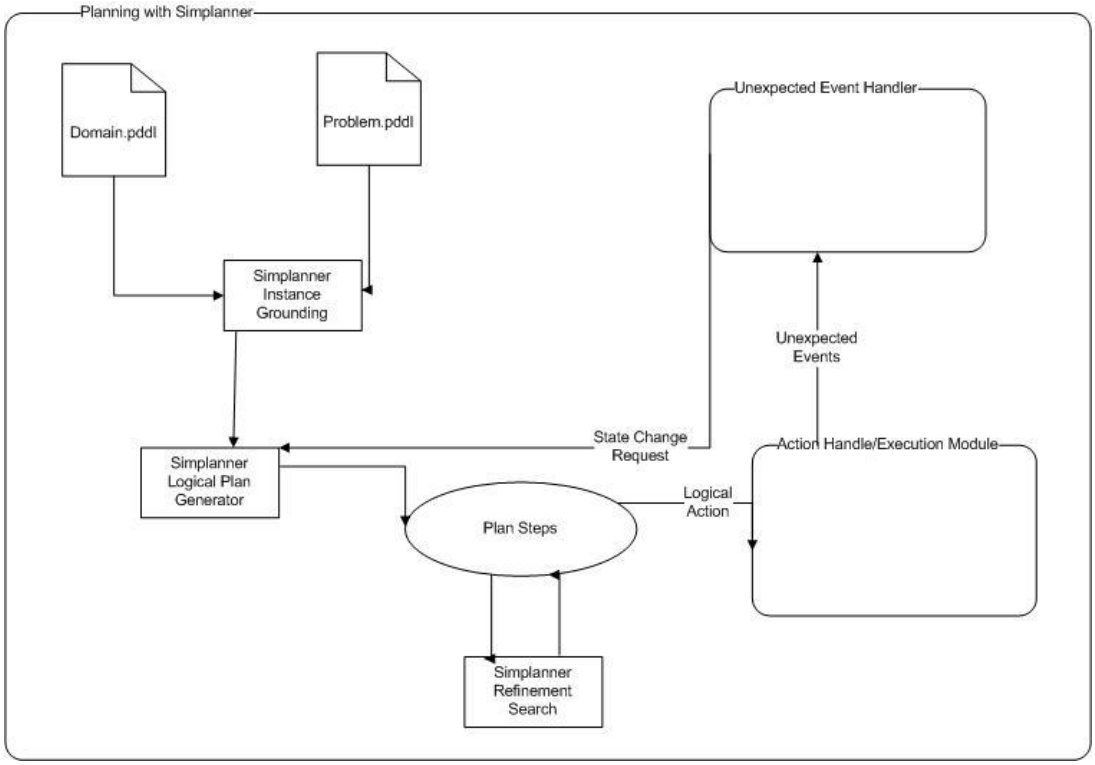

Figure 3-4 Planning Phase

Simplanner is any time planner: it produces a quick logical action and continues to planning as time permits. The planner continuously collaborates with the action handle/execution module and unexpected event handle module. The planner tells a logical action to the action handle/execution module one at a time. During real service execution period that is conducted by the executor, Simplanner continues to operate in order to refine the current plan. During service execution some problems may appear such as information unavailability or service execution failures. In such cases unexpected event handler examines the state and informs the planner about the unexpected situations. Simplanner will then produce a new plan according to the current state.

3.2.3 Action Handling Phase

In this phase, the logical action that is provided by the planner is handled in order to satisfy the user needs. The graphical representation of this phase is given in Figure 3-5.

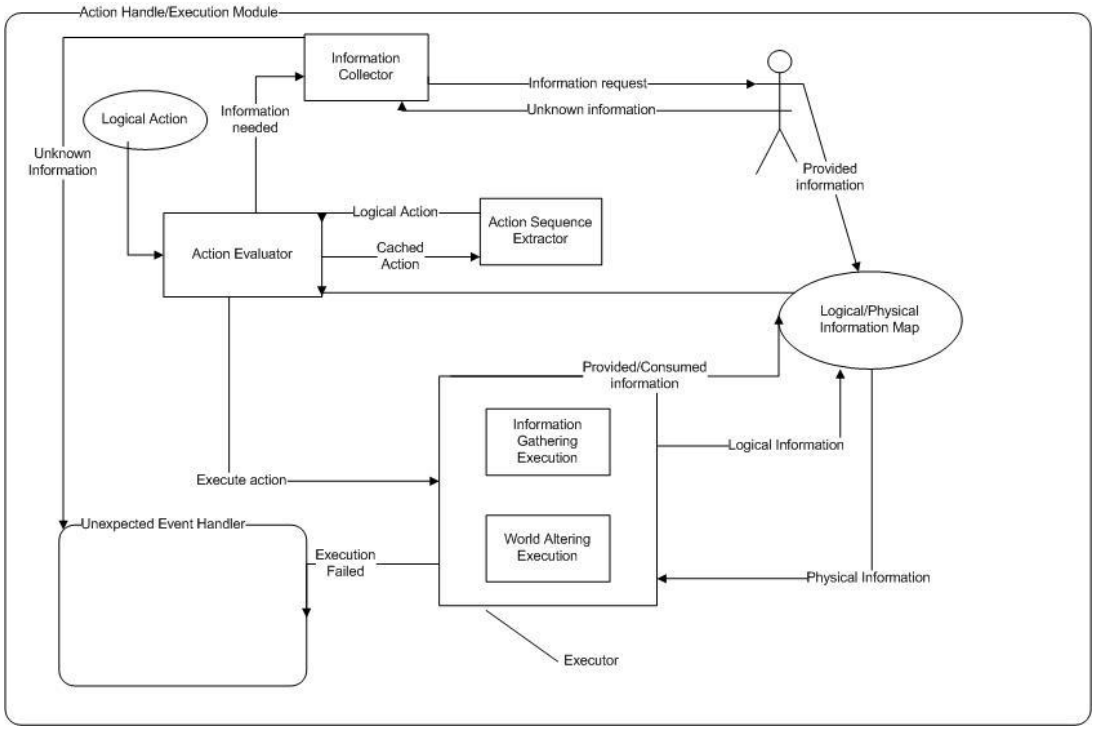

Figure 3-5 Action Handling Phase

Action evaluator initially examines the parameters of a given action and finds the real values of the parameter objects in the logical/physical information mapping. If the information is unknown in the mapping, information needed signal is fired and information collector prompts the user and requests the required information by using the syntactic counterparts of logical parameters that are obtained from WSDL. If the user provides the required information, logical/physical information mapping is updated and the execution continues. Otherwise, information collector sends the

missing information signal to the unexpected event handler which tries to resolve the encountered problem.

After successful termination of a user service request session, the executed action sequence is cached and a new complex action is added to the domain knowledgebase. The precondition of the new complex action is the initial state of the current session and the effect of the new action is the goal state of the current session. If such a cached action is proposed by the planner for future problems, the action sequence extractor gives the saved action sequence and the execution continues with that action list. After the action handling procedure is completed, execution starts.

3.2.4 Execution Phase

In this phase, real service call is performed. Execution handling is done in two ways according to service behavior. If only an information gathering service is executed, it is done as a usual service call. However, if a world altering service is executed, the service is not called directly but indirectly that is conformant with WS-Business Activity and WS-Coordination specifications. The graphical representation of this phase is given in Figure 3-6.

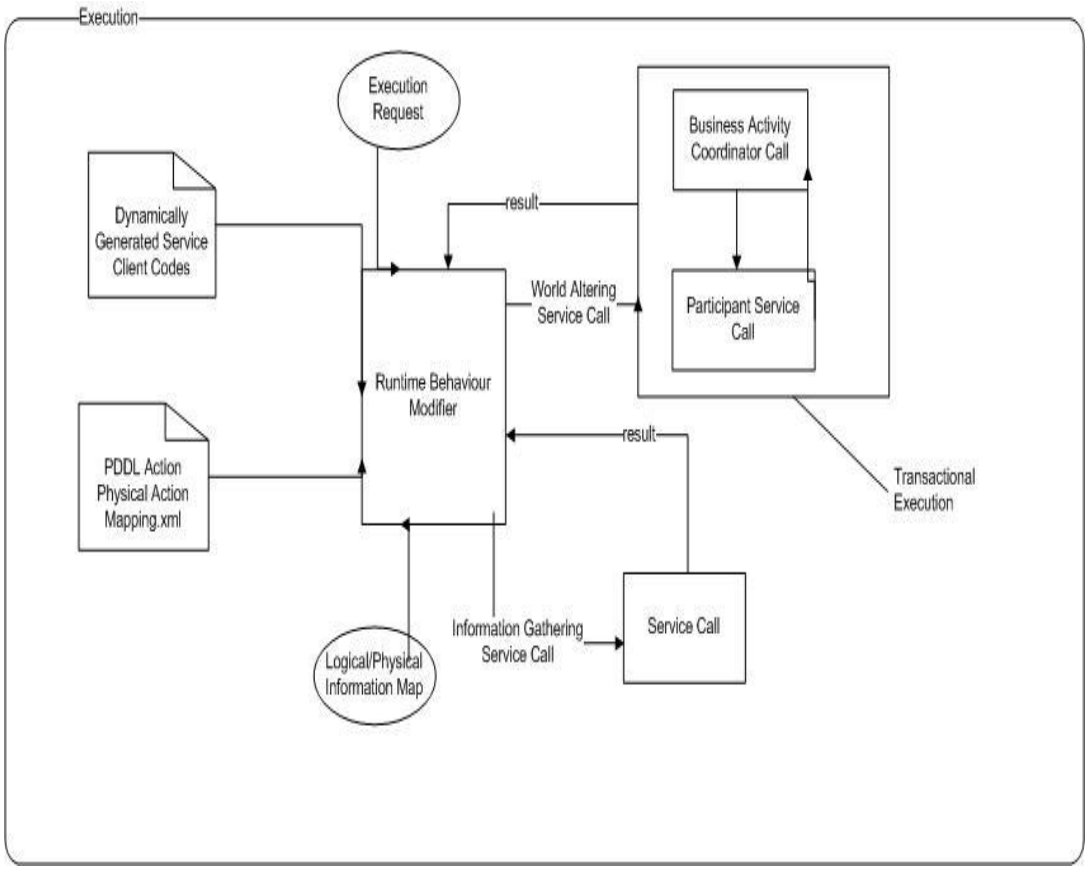

Figure 3-6 Execution Phase

In this phase, dynamically generated service client codes and PDDL action- physical action mappings are used in order to make a real service call. The objects are dynamically constructed at run time by using the mentioned information resources and by the help of reflection mechanism [38]. The real values of service call arguments are collected from the logical/physical information map and the constructed object instances are modified with the collected real data with reflection. If the service provides information, provided information is taken to the logical/physical information map and consumed information is deleted from the logical/physical information map (explained in Chapter 5). If the service to be called has world altering effects, its call is done through business activity coordinator which is generated at the beginning of each session.

3.2.5 Unexpected Event Handling Phase

During service execution and during information collection, some unexpected situations may appear. Service execution may fail because of network problems or wrongly provided arguments or the user may not know the information that is required by the service. Such unexpected situations are handled in this phase. The operations of this phase are described in Figure 3-7.

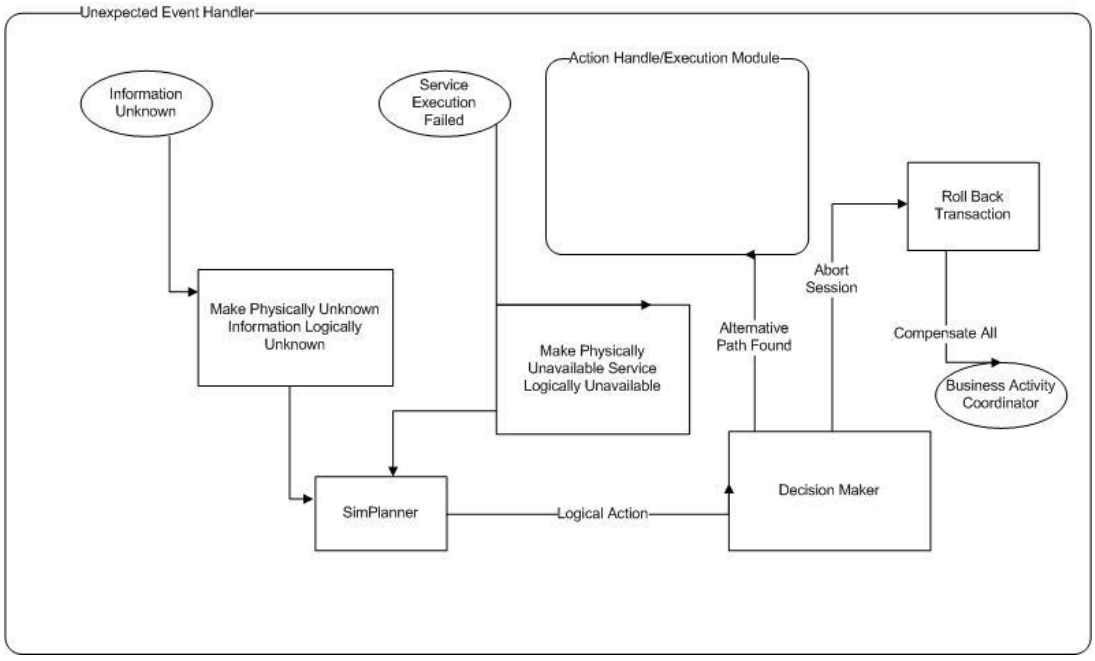

Figure 3-7 Unexpected Event Handling Phase

At the initial state, it is assumed that all services that are specified in the service repository are available for execution and all the required information of services can be provided by the user. If service execution fails, the failed service is logically made unavailable. As a result, the logical action that causes the invocation of the failed service is not considered by the planner in later steps. Planner tries to find alternative services to achieve the user goal. If alternative actions do not exist, no plans can be found to satisfy user needs.

Most of the time, web service requires some input information. If information could not be provided by the user, the logical state that assumes all the required information is going to be provided by the user at runtime is modified. The logical counter of the information which is not known by the user is removed from the state. As a result planner tries to find a service that provides the required information or an alternative path that does not require that particular information. If planner is able to find such a service, execution continues; otherwise the session is terminated.

If replanning cannot produce new ways to achieve goals of the user after the occurrence of unexpected events, the transactional operations are rolled back. The business initiator sends the necessary signal (i.e. compensate) to all participant services through business activity coordinator. This prevents side effects of unsuccessful attempts.

3.3 OWL-S/PDDL Mapping

OWL-S/OWL to PDDL mapping is an important step for automated web service composition with AI planners. Most of the AI planners use PDDL data format for domain and problem representation as in the Simplanner case. On the other hand, semantic web services are described using OWL-S and users can provide their requests using constructs that are defined in domain ontologies which are most probably defined using OWL. Therefore some sort of translation is needed between these distinct formats.

OWL-S/OWL has many common characteristics with PDDL which enables to translate from one format to another straightforwardly. There exist previous works to perform this translation. They all propose similar things with some small distinctions. In works [29], [39], [40] some techniques are proposed for OWL-S/OWSL to PDDL conversion that are very similar to each other. In [39], the most important difference is the usage of KIF language for precondition and effect representation which is one of the recommended languages in OWL-S 1.1 specification. In work [29], a custom language namely PDDXML [29] is used for precondition and effect representation which is acceptable. In OWL-S 1.1

specification, the language that is used for precondition and effect representation is not determined explicitly so custom languages can be used as in [29]. In work [39], input and output parameters of OWL-S actions are directly converted to PDDL parameters. In [29], as well as same conversion a new predicate is generated for PDDL representation, namely agentHasKnowledgeAbout(X) which is used to show information availability. AgentHasKnowledgeAbout(X) construct is necessary for web service composition domain, especially for the cases where information is partially observable. In this thesis, this construct is used to decide about the information gathering resource: is it the user or is it another web service? This thesis accepted the approach that is presented in [29] with some modifications that are needed to handle nondeterministic cases such as service failures. For handling service failures, new predicates are generated for each web service that represent their availability. In this work, for precondition and effect representation, PDDXML is used as in [29], which is easy to use and sufficient for the required information representation. The conversion rules are adapted from [39] and [29] except for the predicates that are used for service availability. The service availability predicates are proposed in this work for handling nondeterministic service executions. The conversion rules between OWL-S/OWL and PDDL are as follows:

- OWL classes are converted to PDDL types, class-subclass hierarchy is preserved during conversion.

OWL Definition	**PDDL Definition**
<owl:Class rdf:ID="Region"/>	(:types Region – object)
<owl:Class rdf:ID="ConsumableThing"/> <owl:Class rdf:ID="PotableLiquid"> <rdfs:subClassOf rdf:resource="#ConsumableThing" /> </owl:Class>	(:types ConsumableThing – object PotableLiquied - ConsumableThing)

Figure 3-8 OWL Class – PDDL Type Mapping

- OWL properties (object properties, data type properties, functional properties...) are converted to PDDL predicates.

OWL Definition	PDDL Definition
<owl:Class rdf:ID="Wine”/> <owl:Class rdf:ID="WineGrape”/> <owl:ObjectProperty rdf:ID="madeFromGrape"> <rdfs:domain rdf:resource="#Wine"/> <rdfs:range rdf:resource="#WineGrape"/> </owl:ObjectProperty>	(:types Wine – object WineGrape – object) (:predicates (madeFromGrape ?WineParameter – Wine ?WineGrapeParameter – WineGrape))

Figure 3-9 OWL Property – PDDL Predicate Mapping

- OWL individuals that are instances of OWL classes are converted to the PDDL objects.

OWL Definition	PDDL Definition
<owl:Class rdf:ID="Region”/> <Region rdf:ID="CentralCoastRegion" />	(:types Region – object) (:objects CentralCoastRegion – Region)

Figure 3-10 OWL Individual – PDDL Object Mapping

- OWL-S service description is converted to PDDL action. The domain.pddl is constructed through OWL-S descriptions. valid[ServiceName] predicate is added to the preconditions of each service.

OWL Definition	PDDL Definition
<service:Service rdf:ID="BookFinderService"> <service:presents rdf:resource="#BookFinderProfile"/> <service:describedBy rdf:resource="#BookFinderProcess"/> <service:supports rdf:resource="#BookFinderGrounding"/> </service:Service>	(:predicates validBookFinderService) (:action BookFinderService (:precondition (validBookFinderService)))

Figure 3-11 OWL-S Service – PDDL Action Mapping

As mentioned before for precondition and effect representation, PDDXML language that is presented in [29] is used. PDDXML [29] is a very simple xml language that uses OWL properties and OWL-S service parameters to describe preconditions and effects.

- PDDXML preconditions are converted to PDDL action precondition.

PDDXML Definition	PDDL Definition
<precondition> <and> <pred name="validPersonalFlightAccount"> <param>?Person</param> <param>?AccountData</param> </pred> </and> </precondition>	(:action ServiceName :parameters (?Person - Person ?AccountData - Account) :precondition (validPersonalFlightAccount ?Person ?AccountData))

Figure 3-12 PDDXML Precondition – PDDL Precondition Mapping

- PDDXML effects are converted to PDDL action effects.

PDDXML Definition	PDDL Definition
<effect> <and> <pred name="isBookedFor"> <param>?Flight</param> <param>?Customer</param> </pred> </and> </effect>	(:action ServiceName :parameters (?Flight - Flight ?Customer - Person) :effect (isBookedFor ?Flight ?Customer))

Figure 3-13 PDDXML Effect – PDDL Effect Mapping

- OWL-S input parameters and output parameters are converted to the PDDL parameters. AgentHasKnowledgeAbout predicate is placed to the preconditions of actions for each input and it is placed to effects part of actions for each output parameter.

OWL-S Definition	PDDL Definition
<profile:hasInput> <process:Input rdf:ID="Flight"> <process:parameterType rdf:datatype=TravelOntology.owl#Flight </process:parameterType> </process:Input> </profile:hasInput>	(:action ServiceName :parameters (?Flight - Flight) :precondition (agentHasKnowledgeAbout ?Flight))
<profile:hasOutput> <process:Output rdf:ID="VehicleTransport"> <process:parameterType rdf:datatype=TravelOntology.owl#Transport </process:parameterType> </process:Output> </profile:hasOutput>	(:action ServiceName :parameters (?VehicleTransport - Transport) :effect (agentHasKnowledgeAbout ?VehicleTransport - Transport))

Figure 3-14 OWL-S Parameter – PDDL Paramter Mapping

- The initial state and goal state of the user request are described in OWL and they are converted to init and goal descriptions of problem PDDL.

In the beginning, it is assumed that all required information can be provided by the user except for the particular information that is included in goal statement that requires information gathering activity. In such a case the user explicitly says that he/she does not know that particular information and intends to learn that information. So, for all defined object instances for which the user does not explicitly state unavailability in problem description, "agentHasKnowledgeAbout (obj)" predicate is added to the initial state for all PDDL objects. All services are assumed to operate initially and for all service definitions "validServiceName" predicates are added to the initial state.

OWL Definition	PDDL Definition
Initial State: <VehicleTransport rdf:ID = "TransportToHospital"/> <Patient rdf:ID = "Patient_0"/> **Goal State:** <VehicleTransport rdf:ID = "TransportToHospital"/> <Patient rdf:ID = "Patient_0"/> <VehicleTransport rdf:resource="#TransportToHospital"> <isBookedFor rdf:resource="#Patient_0"/> </VehicleTransport>	(:objects TransportToHospital – VehicleTransport Patient_0 – Patient) (:init (validService1) (validService2) …… (agentHasKnowledgeAbout TransportToHospital) (agentHasKnowledgeAbout Patient_0) ……) (:goal (and (isBookedFor TransportToHospital Patient_0))))

Figure 3-15 OWL State – PDDL State

In the example above, the user represents his/her request through OWL statements. The user defines logical objects and the desired state about the logical objects. If the user needs information but not a state change, a logical object “obj” is defined as in the example above and “agentHasKnowledgeAbout obj” is added to the goal definition. In such a case “agentHasKnowledgeAbout obj” is removed from the init definition.

CHAPTER 4

AUTOMATED WSC WITH SIMPLANNER

Simplanner is a very suitable planner for web service composition domain; since it provides the ability to handle partially observable, nondeterministic environments in a time efficient manner. The application of Simplanner to the WSC domain is presented in this chapter with a brief introduction to Simplanner algorithm. This chapter concludes with a discussion about the advantages that Simplanner provides for WSC domain.

Simplanner [6] is a kind of domain independent AI planner that is designed to operate on highly dynamic, partially observable environments with time limitations. Simplanner is any time planner, that is, it finds an initial solution to the presented problem very quickly and tries to refine the initial solution as time permits. Simplanner is also an online planner which makes it highly resistant to the unexpected situations: the plan execution can start without a total plan is generated. Lastly it allows modifying the current state to another desired state which enables to deal with incomplete information and unexpected situations. As a result, the required flexibility for real world problems can be obtained.

4.1 Introduction to Simplanner

One of the most important features of Simplanner is its responsiveness in real time. Planning problems are generally very complex problems to solve and their computational complexity is generally exponential so it is very difficult to give a solution in a short period of time.

There exist distinct approaches for producing solutions in a timely manner such as precompiled solutions to the problems and any time planning approaches [6]. In most of the real world cases as in the WSC, using precompiled solutions is out of the question since there exist millions of possible problems and solutions, so what is needed is to use any time approach like Simplanner. By using anytime approach, Simplanner gives an initial solution in polynomial time which is reasonable and it continues to plan up to the execution point. As mentioned before, the execution can start any time before the total plan generation. After each execution step, Simplanner considers the current state and produces a plan for the current situation [6].

Achieving any time planning is a very difficult task and generally requires important amount of domain knowledge as in HTN case that uses domain information for task decomposition. Using HTN based solutions is limited in most of the real world problems since there does not exist sufficient domain knowledge in most of the cases. For instance for WSC domain, such a task decomposition information can be obtained from complex process definitions of OWL-S descriptions but it is very limited. The main aim of those complex process definitions is to show the interaction details for the service requester but not to give information about the domain and in most of the cases such complex interactions are not available. What we need is a domain independent any time planner for time critical operations. The novelty of the Simplanner comes from the fact that it achieves anytime planning without any need to domain specific information.

Figure 4-1 adapted from [6], represents the integrated architecture of planning and execution steps of Simplanner.

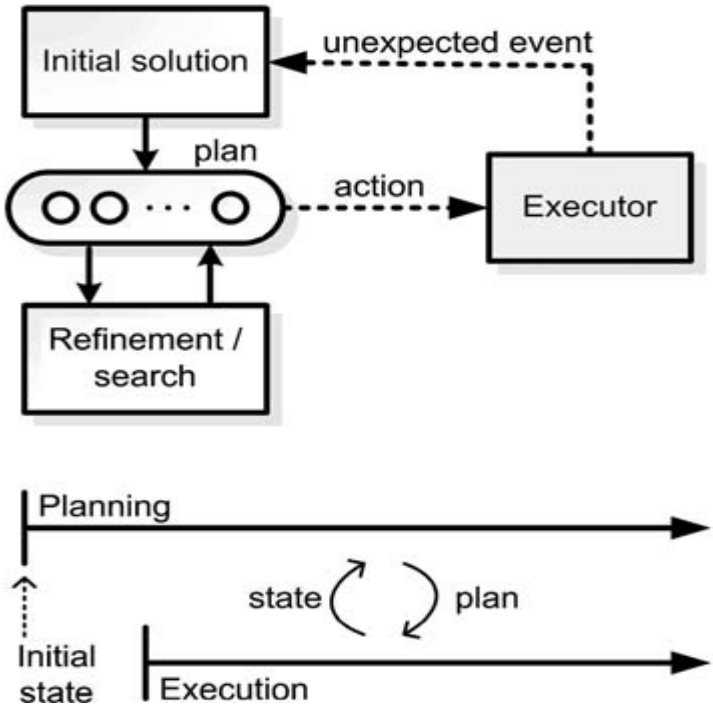

Figure 4-1 System of Simplanner

Simplanner continuously interacts with the custom executor logic. It provides logical high level actions to the executor and gets the information about the unexpected events from the executor. If an unexpected event occurs, planner rejects the current plan and tries to find out a new plan that is suitable for the current state. If everything goes well, planner does not try to find a new plan from scratch but to improve the current plan by searching the state space as time permits [6].

The aim of the Simplanner is to find a complete solution to the presented problem as other AI planners, but Simplanner has also another goal which is very important and provides the main distinction from other planners. Simplanner concentrates on the initial action but not on the whole plan because of the anytime principles. The algorithm of the Simplanner is based on the depth limited heuristic search that can be interrupted at any time. If interruption occurs, planner returns the most promising action that will be used to reach the goal state. Otherwise it continues to provide better solutions [6].

Figure 4-2 that is presented in [6] shows the working mechanism of Simplanner very briefly.

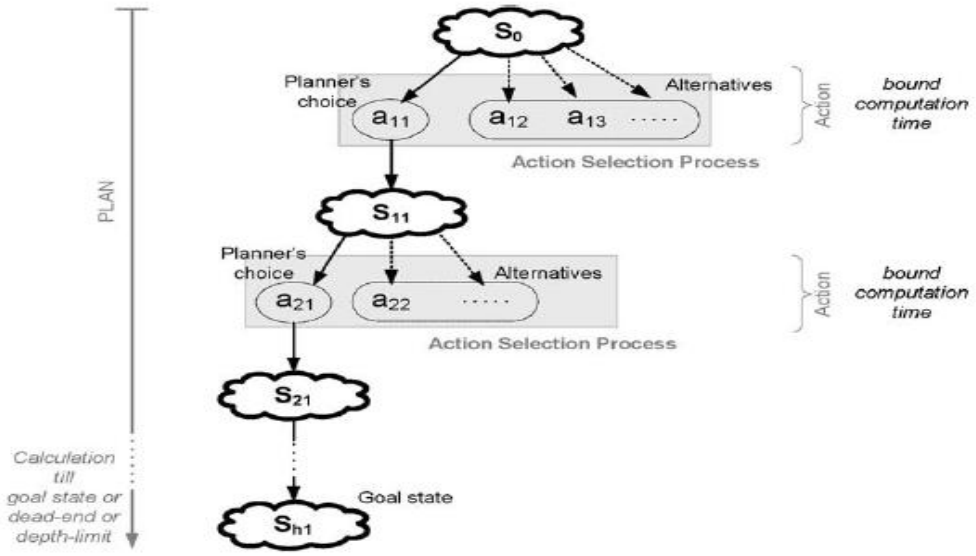

Figure 4-2 Simplanner Working Mechanism

"S_0" represents the initial state. The planner starts to search for the most useful action in the current state for reaching the goal. If the planner is interrupted at any time because of a request for an action, it provides the best action that it has found up to that point and stores the other actions as alternatives. After selecting an action, the planner changes the current state to the state that is resulted from executing the proposed action. The negative effects of the executed actions are removed from the current state and positive effects are added to the current state and the current state is changed to the newly constructed state and search continues [6].

The search continues until the goal state is reached or a depth limit is reached. Even if the goal is reached, the expansion procedure does not terminate. The planner tries to find better solutions for the problem until interrupted by external entities. The expansion procedure is not a backtracking but a back jumping. That is, expansion

continues with the nodes located at higher levels since initial actions are more important for the rest of the plan and those actions are executed at the points that are farthest to the goal state [6].

Due to the characteristics of the general Simplanner algorithm, the most important focus is on producing a promising action at a time, so the most important part of the Simplanner algorithm is its action selection procedure. Simplanner achieves to produce an initial plan very quickly but by losing completeness. It is not guaranteed to find the plan even if it exists. This is an undesired property but it is necessary to work with real world problems in a timely manner.

Simplanner algorithm is based on goal decomposition and searching with heuristics obtained from relaxed planning graph (RPG) [41]. RPG is a rich source of information and it is used by most of the AI planners especially by the domain independent ones.

The algorithm of the Simplanner [6] contains three important steps. Some brief descriptions of these steps are provided below in order to get the idea at least intuitively. The details of the algorithm can be found in [6].

Relaxed Plan Graph generation

In this phase classical RPG is constructed but with some modifications that are used for handling unknown situations. RPG consists of literal and action levels and shows mutex relations between them. By using RPG, one can estimate the distance to the goal. Goal can be obtained on the literal level that does not contain any mutex relations between goal literal pairs. RPG is used as an important information source in other steps of Simplanner algorithm [6].

Subplan Construction

In this phase, goals are decomposed and subplans are generated for achieving each goal in order to find a solution to the problem in polynomial time. Logical statement that is a part of the sub goal with the highest cost is selected to be provided, since repairing such costly cases is difficult in later stages. Cost is determined from the

RPG and after logical statement is selected, action selection procedure starts. In action selection, the action with the lowest cost is selected. Action selection that produces desired logical statement depends on the cost which is obtained from action evaluation function of the Simplanner. This function considers reachability costs of preconditions of action, and some other properties such as negative interactions of the other effects of the action with the other action steps of the subplan [6].

Subplan construction may not be completed when planning is interrupted for an action request. It is guaranteed that, first action of the plan that is found so far is a valid action that can be executed. However, some problems may exist in forthcoming action steps of the plan. As a result the search continues to repair partial plans up to the next interruption of the planner. Figure 4-3 that is obtained from [6] shows graphically the subplan and why it is needed to continue the search process after initial action is provided. Some discovered actions in later steps may destroy the preconditions of the previously discovered actions.

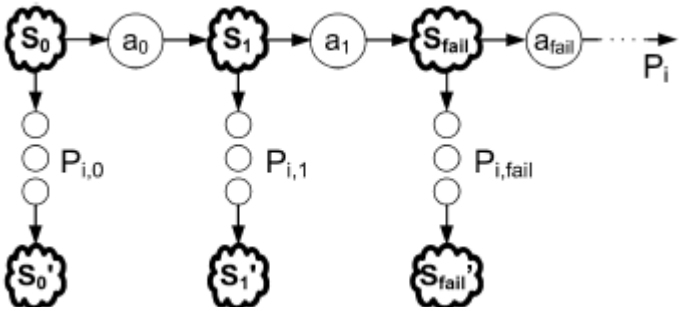

Figure 4-3 State Action Conditions

Here, the state "S_{fail}" that contains the preconditions of action "a_{fail}" is destroyed by prior actions such as "a_0" and "a_1", so subplan should be repaired.

Subplan Ordering

When the planner is interrupted for an action request, subplans should be ordered and the action "a_0" of the first subplan according to the ordering process should be returned. For ordering process, RPG and some loop detection mechanisms are used. By using those resources negative interactions and positive interactions are tried to be found. As a result, the required ordering is conducted among subplans and the first action (a_0) of the subplan that is ordered first is returned to the action requester [6].

Simplanner's paradigm (that is producing one action at a time for problem solution very quickly) is a very important property and it is very usable especially for WSC domain. Experimental results that are presented in [6] also show that it is highly competitive with other domain independent AI planners.

4.2 Simplanner Application to WSC Domain

Some features of Simplanner such as its ability to deal with unexpected situations and the reactivity that it provides are very valuable for WSC domain. In this thesis, these features of Simplanner are used and a service composition agent is constructed that is competitive with the ones that are available in the literature.

Simplanner application and the use of its features are provided by means of four ways. First, service composition problem is stated in PDDL data format at a higher level since planner is working with PDDL data format. Second, service invocation component is integrated with Simplanner through some communication interfaces. Real execution component requests high level logical actions from planner and planner provides them to the execution component. As a third step, the execution component performs some processing and informs the unexpected event handler about the unexpected events. Lastly, an interface is provided between the planner and unexpected event handler component. Handler requests state changes from the planner that are suitable to the current situation. High level architecture is presented in Figure 4-4.

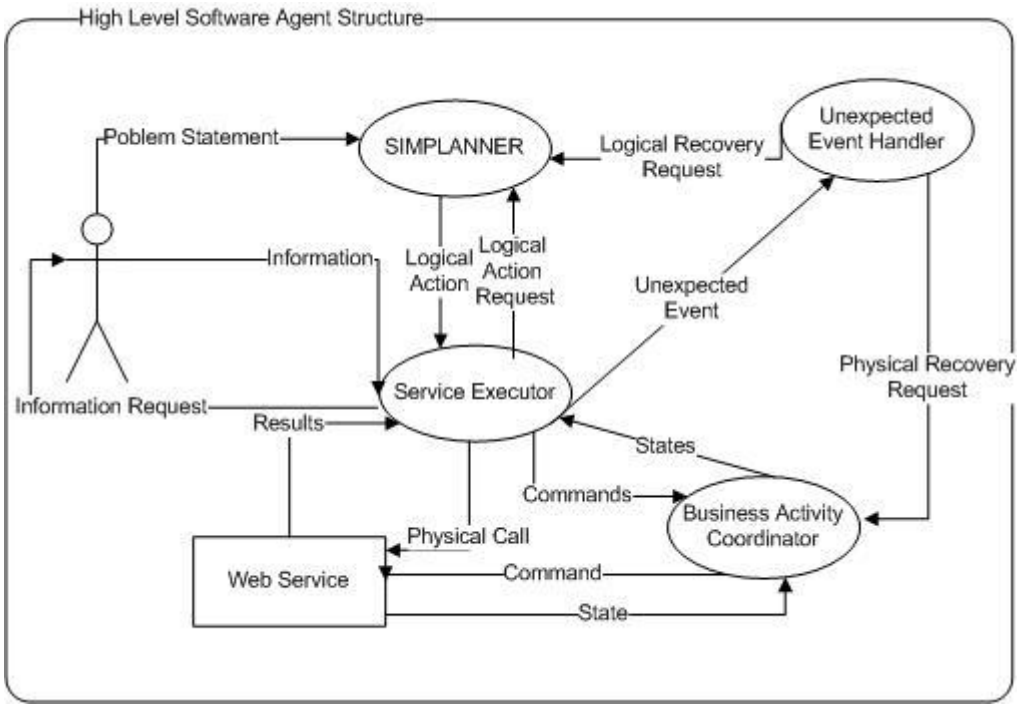

Figure 4-4 High Level Software Agent Architecture

The details of the steps can be described as follows:

4.2.1 Problem Statement to the Simplanner

Simplanner requires information about the domain and about the problem in PDDL data format. The domain information is constructed by using the OWL-S semantic descriptions of web services and domain ontology that is used for describing those services and user's problems. High level logical action definitions as well as type and predicate definitions are constructed using these resources according to the rules described in section 3.3 (OWL-S/OWL to PDDL mapping part).

Action definitions that are constructed during problem statement part are considered high level, logical actions since both actions and their parameters do not exist in the real physical world. Therefore, they cannot be used directly. For instance, consider the PDDL action in Figure 4-5 which is obtained from an OWL-S service description after the required translation is applied in travel domain.

```
(:action BookFlightAtomicProcess
: parameters (?Customer – Person  ?AccountData - Account ?Flight - Flight)
: effect (and (isBookedFor ?Flight ?Customer))
)
```

Figure 4-5 Logical Action Example

"BookFlightAtomicProcess" is considered as a logical action, since it is not an executable action. There is no information for performing the real execution. For instance, service endpoint is not provided: the host and port combination that serves this action is unknown. The communication protocol that is needed to call the service is not known either.

Similar problems exist for parameters, preconditions and effects as well. For instance the type of the parameter "?Customer" is "Person" in the example above, but "Person" type is a semantic type which cannot be used at the execution level. The generated effect is a logical effect as well. Some other physical operations should be done for a real effect.

After the logical domain knowledge is constructed, one needs to present the logical problem to the Simplanner as well. The problem can only be stated logically in this level because of the same reasons mentioned above. The information that is presented to the planner is always some semantic knowledge which is necessary for the planner to operate. Real operations require syntactic knowledge that is handled in later stages.

The problem is provided by the user through logical statements using OWL individual definitions and relations between OWL individuals. Those statements that are represented in ontology web language are translated to the PDDL objects and PDDL initial and goal states according to the rules defined in section 3.3. Both OWL statements and PDDL statements are logical high level constructs and gives

the same information. The difference between the two lies in their syntax; otherwise the meanings of the statements are totally the same. Users provide their requests abstractly and the details of the request are extracted by the software agent in later stages by asking the user the required information. The simple example provided below clarifies the mentioned things here.

Suppose the user wants to make a transportation reservation to the hospital for a person. The user defines his request by using the domain ontology constructs and defines the initial state and the desired state. In the initial state, the statements in Figure 4-6 are provided by the user to show the logical availability of a particular person and a particular transportation.

```
<Person rdf:ID="Patient_0"/>

<VehicleTransport rdf:ID="TransportToHospital"/>
```

Figure 4-6 Example Initial State

In the goal statement, the user shows the availability of the same logical objects and presents the desired relation between these two logical objects. They are all logical; there is no information about the details of a particular transportation and particular person. In this level, the only useful information is semantic types of objects which are sufficient in this phase.

```
<Person rdf:ID="Patient_0"/>
<VehicleTransport rdf:ID="TransportToHospital"/>
<VehicleTransport rdf:resource="#TransportToHospital">
        <isBookedFor rdf:resource="#Patient_0"/>
 </VehicleTransport>
```

Figure 4-7 Example Goal State

OWL statements in Figure 4-7 describe that, there is a desire to book a thing whose semantic meaning is “VehicleTransport” to another thing whose semantic meaning is “Person”. The semantic details are going to be obtained by the actions proposed by the planner and syntactic details are going to be obtained by the service composer agent by asking questions to the user.

Sometimes users do not request a change in the world state but they only need to find out information. In such cases, agentHasKnowledge(X) construct of [29] is used. In this case, users provide limited information about their requests. For instance, suppose the user wants to find out the flight number of a particular flight. In such a case he/she presents the statements in Figure 4-8 to the system initially.

```
Init:
 <Flight rdf:ID="Flight"/>
Goal:
<Flight rdf:ID="Flight"/>
<agentHasKnowledgeAbout rdf:resource=”#Flight”>
```

Figure 4-8 Information Gathering Request Example

The details of the flight such as its source and destination locations and arrival time are all unknown at this stage. The system only knows that user wants to retrieve the flight number of a particular flight.

It is possible to collect more details about the problem at the initial stage, but this is an ineffective way. The user cannot know all of the required information that will be needed in later stages. In such a scenario the user should provide too much information, important amount of which is not used most probably. As a result that solution is unacceptable. In our work, we propose to request only necessary information from the user after understanding their problem in a higher level.

4.2.2 Planner – Execution Component Integration

After high level problem and domain information are presented to the planner, it tries to find out a logical solution to the current problem. Two important assumptions are made at the beginning of planning. First, all the actions that are available in the domain knowledge base are executable. Second, all the necessary information that will be required in later stages can be provided by the user. These assumptions are the initial assumptions. The validity of these assumptions is determined after some interactions with the user and web services. If the assumptions are wrong, they are handled easily by the features of Simplanner.

The initial assumptions mentioned above are asserted to the planner through the use of two predicates: “validService” and “agentHasKnowledgeAbout”. Each logical action definition that corresponds to a physical web service operation includes statements with these predicates in their precondition part. In order to execute any service, it should be physically available and the parameters that are required for executing that service should be known apriori. Consider the example in Figure 4-9:

```
(:action BookFlightService
: parameters ( ?Customer - Person ?AccountData - Account ?Flight - Flight)
: precondition (and (validBookFlightService)
(agentHasKnowledgeAbout ?Customer)
(agentHasKnowledgeAbout ?AccountData)
(agentHasKnowledgeAbout ?Flight)
)
```

Figure 4-9 Asserted Service Preconditions

"BookFlightService" is the logical counterpart of the service operation that does booking operation. The service requires three parameters, the semantic types of which are "Person", "Account" and "Flight" respectively. The precondition statements show the mentioned idea. The parameters should be known and service should be available.

The action definitions provided in the domain.pddl are generic action definitions, so they cannot be used directly even logically. Simplanner does a grounding operation on actions with the defined pddl objects in problem.pddl before searching for a plan. The result of grounding process is real logical actions. For instance, for the logical instantiation of the action "BookFlightService", three logical objects with the required types are necessary. Suppose such objects are defined in the problem.pddl with the corresponding types as in Figure 4-10:

```
(: objects Person1 - Person  Person2 – Person  Account1 – Account Flight1 – Flight)
```

Figure 4-10 Example Logical Objects

The logical action definitions that are obtained after action instantiation of the "BookFlightService" are as follows:

```
BookFlightService Person1 Account1 Flight1
BookFlightService Person2 Account1 Flight1
```

Figure 4-11 Example Grounded Actions

At the initial problem statement, the user defines some logical objects and represents the relationship between them, but some other objects with other types may be necessary to solve the problem. Suppose the user wants to do a booking and represents his/her problem as described in "Problem Statement to the Simplanner" part. Initially the user does not know that "BookFlightService" is going to be used for the solution of the problem, so he/she does not know the required parameters of the service either. As a result, some of the required logical objects that are necessary for the problem solution are not provided by the user. This is an important issue. For the "BookFlightService" example, the logical service instantiation needs some "Account" typed object. If it is not defined by the user in the problem statement, the service cannot be used even if it is required for the solution. That is a problem and in this work the solution to that problem is provided by constructing one logical object corresponding to each PDDL type if not constructed by the user explicitly.

The instantiation of any action becomes possible after logical object construction with each PDDL type but the instantiation is not sufficient for logical execution. As mentioned before, each logical object that is used as a parameter by the logical action should be converted to the physical ones. Before the physical execution, the software agent asks for the values of those logical objects and acts according to the answer.

Considering the action “BookFlightService”, the statements in Figure 4-12 are not available in problem.pddl before the planning procedure starts. They are constructed by the software agent after examining the user request.

```
(:objects  PersonObj - Person  AccountObj – Account FlightObj – Flight)
(:init (and
(validBookFlightService)
(agentHasKnowledgeAbout PersonObj)
(agentHasKnowledgeAbout AccountObj)
(agentHasKnowledgeAbout FlightObj)
))
```

Figure 4-12 Initial Logical Statements Example

If some logical objects are constructed with types “Person”, “Account” or “Flight” in the user request, another object with the same type will not be constructed again. As mentioned before the pddl objects are logical objects, so same objects can be used for instantiation of distinct actions. The real values of these logical objects are obtained at run time so the use of the same logical object by distinct services does not mean that distinct services are called with the same arguments. Suppose “service1” uses “obj1” as a parameter and “service2” uses “obj1” as a parameter again. During “service1” call, the real value of “obj1” is requested from the user or from another service the details of which are described later. During “service2” call, the same procedure is applied from scratch. Therefore, distinct arguments are used for the same logical objects. In conclusion, one logical object for each pddl type is sufficient and they do not cause confusion during real service call since physical values obtained in later stages.

At the user request examination step of the software agent, information gathering requests are determined. If such a request is available, the software agent understands that the user does not have any information about the logical object that is mentioned in the request statement about the information gathering activity. The software agent then does not make the assumption that user can provide information about the particular logical objects and removes those particular objects from the list of known objects. For instance, if the user presents an information gathering request as in Figure 4-8; the software agent will not construct an object with type "Flight" and it does not add "agentHasKnowledgeAbout Flight1" statement to the init section of the problem pddl.

After the initial problem is given to the planner, the planner tries to find a solution. The execution module requests an action from the planner after some deliberation time. When the request comes from the executor, the planner returns the most promising logical action (that is grounded action with logical objects) to the executor. While the executor is doing its own job (real service execution, information collection from user, etc…), the planner continues to search for a better solution and repair any problems that may occur in later steps of the proposed initial solution. The planning procedure continues until an interruption by the executor again.

4.2.3 Execution Component and Its Integration with Unexpected Event Handler Component

After the execution component retrieves the logical action to be executed from the planner, its turn starts. As mentioned before, logical actions are not complete and not executable. Some processing is needed for making them executable.

Real service execution is described in Chapter 5 in detail. Before real service execution, some more steps are needed. Action handler component which has very important function in the whole system does its processing. It prepares logical actions for execution in two ways.

First, it tries to collect real parameters that are physical counterparts of logical action parameters from the user or from other web services. For instance, the planner provides a logical action like in Figure 4-13:

```
BookFlightService Person1 Account1 Flight1
```

Figure 4-13 Logical Action Example

“Person1”, “Account1” and “Flight1” are the objects that are defined in problem.pdll description as discussed earlier. The only information about these logical objects is their semantic types at this stage, which is not usable for real service execution.

Logical/Physical map, which is an in memory information holder is the core of the mechanism that associates logical entities with their physical counterparts. During the beginning of each session, a fresh Logical/Physical map is constructed (the details are presented in Chapter 5). The Logical/Physical map contains syntactic counterparts of semantic objects and their current values. Syntactic counterparts of semantic objects are obtained by processing the grounding part of OWL-S service descriptions and WSDL descriptions of services. Initially all real values of the syntactic counters of logical entities are unknown. During the processing, those values are obtained from the user or from other services.

When a logical action is presented to the action handler, it extracts the parameters from the logical action and it looks up to the Logical/Physical map to find out their real values. If Logical/Physical map is able to provide that information, the action handler directs the action to the real executor. If the answer “unknown” is returned from the Logical/Physical map, information collector’s turn starts. Initially, the information collector gets the syntactic details of logical objects from the Logical/Physical map. For instance, a logical object “Request” whose semantic type

is “RequestParameters” is contained in the parameters of the current action definition. The WSDL counterpart of “RequestParameters” that is obtained from OWL-S grounding section is “RequestInfo” type. It has a complex type definition in the WSDL. Suppose “RequestInfo” complex type contains the parts that are presented in Figure 4-14.

DestinationLocation – xsd:string

SourceLocation – xsd:string

ArrivalTime – xsd:dateTime

Figure 4-14 Example Complex Type Parts

When the information collector requests the details of “Request” object, Logical/Physical map provides all the syntactical details and the information collector asks the user for the values of “DestinationLocation”, “SourceLocation” and “ArrivalTime”. The user will provide “DestinationLocation”, “SourceLocation” and “ArrivalTime” for the example above. If the user could not provide the requested information, some other strategy should be used. As an example, suppose the user tries to book a flight and for booking a flight a flight number is needed by the booking service. The above procedure is applied by the software agent and the value of the flight number is asked to the user. If the user does not know the value, he/she tells the software agent that the value is not known. In this case an unexpected situation signal is fired. The details of unexpected event are sent to the Unexpected Event Handler Module.

The second role of Action Handler is to extract the details of cached actions. Action caching is a mechanism for making use of previous experiences in order to speed up the system. (Action caching is described in chapter 5 in detail). When a cached action is proposed by the planner, the procedure that is used for non-cached actions cannot be applied. There does not exist any information about the details of cached

actions in Logical/Physical Map and the details of the actions are obtained from other resources, details of which can be found in Chapter 5.

After the action handler performs its function, the real executor does the remaining steps for calling the web service which is described in Chapter 5. When the real execution is conducted some problems may arise because of network problems or other external reasons. In such cases the executor informs the the Unexpected Event Handler about the unexpected situation and completes its work about that particular action.

4.2.4 Unexpected Event Handler Component and Its Integration with Planner

The executor component sends two kinds of unexpected events to the Unexpected Event Handler: service unavailability and unknown information. The Unexpected Event Handler tries to solve these problems in collaboration with the planner.

The Unexpected Event Handler keeps the logical state that is valid before the service execution. If some problem occurs, it uses this saved state and the information returned from the executor about the problem for constructing a new state that describes the current situation. After a new logical state is constructed, it is presented to the planner and the planner starts to find a solution according to the current situation. Simplanner allows state changes at any time. After a state change request is taken by the planner, the logical actions that it provides to the executor are conformant with the current situation afterwards.

When the service unavailability message comes from the executor to the unexpected event handler, it requests a state change from the planner. The new state is constructed by removing “validServiceName” predicate from the last saved state. For instance if there is a problem during the execution of the booking service “BookFlightService”, the unexpected event handler removes the “validBookFlightService” predicate from the newly constructed state and informs the planner. After that point, “BookFlightService” cannot be used, since its precondition “validBookFlightService” is not satisfied. The planner will not consider this logical action again and it tries to find other ways for solving the

problem. There may exist alternative ways to solve the same problem. The planner constructs a new solution with alternative services if available. Otherwise, the session is terminated unsuccessfully. If session is terminated without totally completed, there may exist some side effects in the environment because of the previously executed world altering services. These side effects are undesired and are not acceptable in most of the cases. In this work, transactional execution is used for world altering services. If session is uncompleted because of unexpected events, side effects are compensated by the used transaction mechanism. The session termination condition and used transactional mechanism are described in Chapter 6 in detail.

The other unexpected event is information unavailability. The information collector component of the execution module asks the user the values of physical counterparts of logical objects as described before. If the user could not provide the requested information, the unexpected event handler changes the logical state for stating the planner that the particular logical information is not known by the user. Suppose there is a logical object “Flight1” which has a semantic type of Flight. Initially, the logical state contains “agentHasKnowledgeAbout Flight1” predicate which assures the user can provide the required information about logical object “Flight1”. If the information collector cannot get the necessary information about “Flight1” from the user, it tells the situation to the unexpected event handler. The unexpected event handler removes the “agentHasKnowledgeAbout Flight1” predicate from the last saved state and tells the state change request to the planner. After that state change, the planner has two alternatives. First, it can search for other actions that do not require the unknown information and that can be used for problem solution. Second, it can try to find other services that provide the required information. Suppose an action, partial definition of which is presented in Figure 4-15 exists:

```
(:action ProposeFlight
(:parameters ( ……..        ?Flight – Flight)
(:effect  (and ……..…
(agentHasKnowledgeAbout ?Flight)))
```

Figure 4-15 Example Action Definition

This action is grounded with “Flight1” during Simplanner action grounding procedure and it can provide the required information about the logical object “Flight1”. If the planner proposes this information gathering action to the executor, the executor does the real service call. The physical counterpart of the logical object “Flight1” is constructed with the reflection mechanism (details are provided in Chapter 5). Then, real values of these physical counterparts are updated in Logical/Physical map. When an action that requires the information about “Flight1” is proposed by the planner again, the action handler can get the real values that are required for the service call from the Logical/Physical map without any need to ask the user again. An important issue about the Logical/Physical map is that, if real values of the physical counterparts of logical objects that are provided by either the user or by another web service are once used, those real values are cleared from the map even if they are required in later steps because of the reasons described in Logical/Physical map section of Chapter 5.

The details of the information collection procedure from services and users, and Logical/Physical map are presented in Chapter 5 and the details of session termination and the transactional mechanism are described in Chapter 6.

4.3 Advantages of Using Simplanner for WSC Domain

As it can be understood from the discussions above, Simplanner is very suitable for the automated web service composition problem. It’s anytime planning mechanism and its feature that enables starting execution before complete plan generation and

its domain independent working principle are very valuable for service composition problem.

Some important surveys about the behaviors of AI planners are conducted as described in Chapter 2 and some important criteria are determined to evaluate the success of AI planners in this domain. In Chapter 2, the importance of Simplanner is shown for this domain when compared with the other AI planners according to those evaluation criteria.

The most important advantages that usage of Simplanner provides in automated web service composition domain can be summarized as follows:

- Timely response

  In web service composition domain, timely response is very important. Humans present their request to the system and wait for a quick response. Solving planning problems is a very difficult task and generally requires exponential computation time which is not reasonable for this domain. As mentioned before, Simplanner finds a very quick initial solution and tries to refine the problem for finding more optimal solutions. For instance a three step plan is more optimal than a five step plan for the same problem but if finding the three step plan requires hours while the five step plan can be found in seconds, finding five step plan is much more desirable for this domain.

  Simplanner provides timely response that the user of the system does not feel the passing time. If the domain that is worked on is too big, the planner cannot provide timely responses but this is the problem of all domain independent planners. The scalability issues about Simplanner and what "too big" means is discussed in Chapter 7.

- Dealing with Nondeterminism Effectively

  One of the most important advantages that Simplanner provides is its ability to deal with unexpected situations. Web is a very dynamic environment and classical techniques that assume everything is observable and the results of each

action are known apriori is out of question. Simplanner interleaves planning and execution, which provides an important amount of flexibility. Some unknown things and nondeterministic executions can be determined by performing some real operations. Through state change requests, the collected information is presented to the planner which is used for later decisions by the planner. Some other planners provide effective solutions for dealing with nondeterministic cases, but most of them consider all possible situations before execution which is not usable when there exist too many things to consider.

Dealing with nondeterminism at run time is very important if there exist too many unknown things as in WSC case. Dealing with nondeterminism is not sufficient but that operation should be conducted in a short period of time in order not to lose responsiveness. The Figure 4-16 that is taken from [6] shows the effectiveness of Simplanner for dealing with unexpected situations.

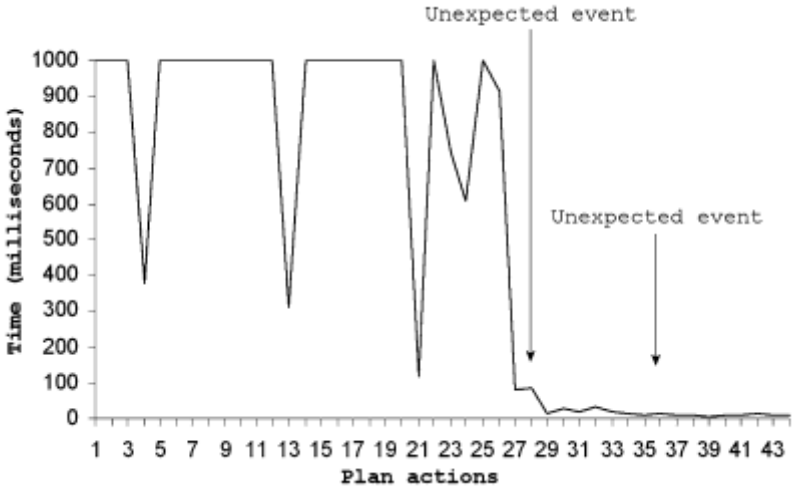

Figure 4-16 Simplanner Behaviour to Unexpected Events

Figure 4-16 shows the required time for producing an action. At initial steps, since the goal is far away, deliberation time is more with respect to the final

steps. As can be seen from the figure above, unexpected situations do not increase the action production time, so dealing with unexpected situations can be done in a responsive way.

- Parallel Execution and Planning

As Simplanner allows interleaving and planning, the execution and plan refinement procedure are done totally in parallel. The deliberation time that is provided to the planner is the real service execution time. During service execution, the planner continues its process for finding a solution. As a result, the planning time has almost no time cost for the system.

- Domain Independence

Dealing with complex planning problems in a timely manner generally requires important amount of knowledge about the worked domain as in the case of HTN planners. In WSC domain, the only information available is OWL-S service descriptions and the only extra knowledge that can be found about the services are provided in the process definition part where the interaction details to the web services are provided. However, that information is very limited and does not provide any more knowledge than the information presented in profile part for most of the cases. Producing solutions very fast without using any domain knowledge is a very important property for WSC domain. Simplanner provides that important feature.

CHAPTER 5

AUTOMATED SERVICE INVOCATION ISSUES

Automated web service invocation module is one of the core components of the proposed framework. Service invocation module enables dynamic service calls without any need to human intervention. Service invocation mechanism uses on the shelf technologies such as java reflection mechanism [38], web service invocation framework (WSIF) [42], WSDL2JAVA tool of Apache Axis [43] framework and integrate them in an efficient manner for achieving automated service call. During service invocation, the arguments that are needed for the service calls are obtained from the in memory data structure namely Logical/Physical map. This data structure provides the communication between the service invocation mechanism and the user provided or service provided input values. After the termination of each successful session, the system caches the action sequence that is used for handling the current problem, in order to use them directly when a similar problem comes.

Section 5.1 presents the mechanism that is used for automated service invocation, section 5.2 describes the Logical/Physical map and section 5.3 describes the action caching mechanism.

5.1 Automated Service Invocation

After the planner provides a logical action to the execution module of the system, the action handler prepares the logical action for real service invocation with the help of information collector and unexpected event handler which are described in Chapter 3 and Chapter 4 in detail.

Action handler makes sure that, the parameters of logical actions have desired syntactic counterparts in logical/physical map, so real service executor can get the required service argument values from the logical/physical map. If the required arguments cannot be supplied by the user or by another service, the planner tries to find an alternative path. If an alternative path could not be disvovered, the session is terminated before reaching the automatic service invocation phase.

There exist two kinds of real service executions in the system. One is information gathering service call and the other is world altering service call. If a service has both information gathering and world altering effects, it is considered as a world altering service call. The difference between them can be understood from their names: world altering service call provides side effects so in order not to change the world in an unintended way, one makes sure that their call should be made transactional. The mechanisms that are used to call these two kinds of services are different. The information gathering services are called as usual but the world altering ones are called by conforming to the WS-Business Activity and WS-Coordination standards [7], [12]. The details about transactional calls and the required information about the web service transaction frameworks are presented in Chapter 6.

Some preprocessing is conducted for automated web service invocation purposes as well. Simplanner uses the semantic knowledge extracted from OWL-S service descriptions but the real service executor needs syntactic information to operate. The syntactic information is collected from WSDL descriptions of the used web services. WSDL contains all details for service calls such as syntactic types of operation arguments, service end point, required communication protocol, etc.

For a real service call, some machine codes are required which are produced automatically by using WSDL2JAVA tool of Apache Axis [43] framework in this thesis. WSDL2JAVA tool produces all required service client codes and the implementations of all defined complex types in WSDL. Service client implementations are compiled and the metadata about the compiled codes are stored to the PDDL Action - Physical Action mapping.xml file. This file holds the

information about the real implementations of logical actions and their parameters that are produced through the mechanism described above. For constructing PDDL Action - Physical Action mapping.xml file, it is not sufficient to use only the information that is obtained from client stub generator. The mapping between the logical actions and logical parameters and their physical counterparts are also required. This mapping information is obtained from the grounding section of OWL-S semantic service descriptions.

In order to extract the relationship between semantic types and syntactic types (that is, to associate logical actions and their logical parameters with their physical counterparts), the grounding extractor processes the grounding part of OWL-S descriptions. It produces OWLS-WSDL mapping.xml file which contains the same information as the grounding part of OWLS but it is a much easier version to be processed by the software agent. PDDL Action - Physical Action mapping.xml file is constructed by using the OWLS-WSDL mapping and the metadata of the dynamically generated code. It holds all the required metadata for a real service call.

The steps that are described above can be explained in a more concrete way as follows:

- The grounding extractor constructs “OWLS-WSDL mapping.xml” file that associates logical objects and their physical counterparts.

The parts of the grounding section that are processed in this system are the following [2]:

- **wsdlDocument:** Represents the URI of the WSDL.
- **wsdlOperation**: Represents the URI of the WSDL operation, only atomic processes are considered in this work.
- **wsdlInputMessage:** Represents the URI of the WSDL message definition that carries the inputs of the process.

- **wsdlInput:** This part represents the mapping between OWLS input parameters and WSDL counterparts. If WSDL counterparts contain complex definitions, such as, user defined custom types with sequence, choice, etc. constructs, XSLT transformations are generally used for conducting the real mapping. Although such XSLT scripts are required, they are not available in most cases since semantic descriptions are rare currently and the available ones that are constructed for research purposes do not contain those scripts. So the alternative way is adapted in this work. The details of complex types are extracted by parsing the WSDL file and stored in the "OWLS-WSDL mapping.xml" file for associating OWL-S and WSDL types even if there exist complex type mappings without XSLT transformation scripts.

- **wsdlOutputMessage:** Represents the URI of the WSDL message definition that carries the outputs of the process.

- **wsdlOutput:** This part represents the mapping between OWLS output parameters and their WSDL counterparts. The mentioned issues about wsdlInput are valid for wsdlOutput too.

The format of the xml file that is produced by the grounding extractor is as shown in Figure 5-1.

```
<actions>
<action name= "LogicalOperation" wsdlOperation="PhysicalOperation" />
        <inputs>
                <input wsdlName = "LogicalInput" owlsName = "PhysicalInput" />
                .....
        </inputs>
        <outputs>
            <output wsdlName = "LogicalOutput" owlsName = "PhysicalOutput" />
                        …….
        </outputs>
</action>
                ……
</actions>
```

Figure 5-1 Grounded Actions XML Structure

The produced information does not contain any other information than the information provided in OWL-S file, but this information is more easily processible for upcoming phases.

- The client stub generator produces the required machine codes for web service clients by using the WSDL2JAVA tool of Apache Axis framework with the required parameters.

During client generation, the schemas that are required for transactional aspects are presented to Axis. Apache Kandula [44] is used for transactional service execution, details of which are described in Chapter 6. The script that is used by the client stub generator is shown in Figure 5-2.

```
Java  org.apache.axis.wsdl.WSDL2Java -s
-Nhttp://schemas.xmlsoap.org/ws/2004/08/addressing= org.apache.axis.message.addressing
-Nurn:test=test.ceng.metu.edu.tr
-Nhttp://schemas.xmlsoap.org/ws/2004/08/addressing= org.apache.axis.message.addressing
-Nhttp://schemas.xmlsoap.org/ws/2004/10/wsba=org.apache.kandula.wsba
-Nhttp://schemas.xmlsoap.org/ws/2004/10/wscoor=org.apache.kandula.wscoor
-xhttp://wsba.kandula.apache.org
-xhttp://wscoor.kandula.apache.org
-xhttp://schemas.xmlsoap.org/ws/2004/10/wsba
-xhttp://schemas.xmlsoap.org/ws/2004/08/addressing
-xhttp://schemas.xmlsoap.org/ws/2004/10/wscoor   WSDLURI
```

Figure 5-2 Client Stub Generation Command

After machine codes of client stubs are generated, the mapping between the logical action and logical parameters and their physical implementations are constructed. The information that is produced by the grounding extractor is used in this step. In addition, parsing WSDL for extracting details about complex types is conducted at

this stage. The output of this phase is “PDDL Action- Physical Action mapping.xml” file.

The structure of this file is shown in Figure 5-3.

```
<actions><action name=”LogicalOperation” class=”ImplementationClass” endpoint=
“ServiceEndPoint”>
<inputs>
     <input name=”Input” class=”ImplementationClass”>
        <subtype name=”SubType” class=”ImplementationClass”/>
        …….
      </input>
     ……
</inputs>
 <outputs>
     <output name=”Output” class=”ImplementationClass”>
       <subtype name=”SubType” class=”ImplementationClass”/>
                …….

      </output>
       ……

</outputs>
</action>
     ……
</actions>
```

Figure 5-3 PDDL Action – Physical Action Mapping XML Structure

The stated information that is prepared in preprocessing stage is used during real service call that is described later in this chapter. As an example, consider a service which has a WSDL definition as follows (only the relevant parts of WSDL are presented):

```
<definitions name="RequestTransport"
targetNamespace="http://ceng.metu.edu.tr/services/wsdl/RequestTransport"
<types>
<element name = "TransportParameters">
      <complexType>
       <sequence>
        <element name="DepartureLocation" type="xsd:token" />
        <element name="ArrivalLocation" type="xsd:token" />
        <element name="Vehicle" type="xsd:token" />
       </sequence>
     </complexType>
   </element>
   </schema>
  </types>
  <message name="RequestTransportInputMsg">
   <part name="transportparameters" element="tns:TransportParameters" />
  </message>
  <message name="RequestTransportOutputMsg">
   <part name="transportid" type="xsd:int" />
  </message>
  <portType name="RequestTransportPortType">
   <operation name="RequestTransportOperation">
   <input message="tns:RequestTransportInputMsg" name="RequestTransportInput" />
<output message="tns:RequestTransportOutputMsg" name="RequestTransportOutput" />
   </operation>
  </portType>
  <service name="RequestTransportService">
   <port name="RequestTransportPort" binding="tns:RequestTransportSoapBinding">
    <soap:address location="http://localhost:8181/axis/services/RequestTransportPort" />
   </port>
  </service>
```

Figure 5-4 Relevant WSDL parts of Example Service

After parsing WSDL file, the information about the service end point, operation URL, details of type information are obtained and put into the xml file as follows for future uses.

```
<action name="RequestTransportOperation"
class="tr.edu.metu.ceng.wsdl.RequestTransport.RequestTransportSoapBindingStub"
endPoint="http://localhost:8181/axis/services/RequestTransportPort">
<inputs>
<input name="transportparameters"
class="tr.edu.metu.ceng.wsdl.RequestTransport.TransportParameters">
  <subtype name="DepartureLocation" class="org.apache.axis.types.Token" />
  <subtype name="ArrivalLocation" class="org.apache.axis.types.Token" />
  <subtype name="Vehicle" class="org.apache.axis.types.Token" />
  </input>
  </inputs>
  <outputs>
  <output name="transportid" class="int" />
  </outputs>
</action>
```

Figure 5-5 Example PDDL Action – Physical Action Mapping.xml

The class values represent the compiled machine codes that are obtained by the client stub generator. The primitive types such as “int, double” does not have custom implementation classes as “TransportParameters” as expected since they are built in types and their handling mechanism is different during run time which is described later in this chapter.

- Real service call is done.

The steps that are described up to now are all preprocessing steps. The component that makes the real service call uses all the metadata and all the machine codes that are produced previously.

Since the services that will be called and the real values of the parameters that are needed for calling the services are not known before computation begins (but are determined during run time), a mechanism is needed that provides the ability to change run time behavior of applications. Such a mechanism is provided by java

reflection infrastructure [38] and it is extensively used in this work for real service invocation.

After some logical action with its logical parameters is provided to the executor by the planner and the action handler does its processing, the real service invoker starts execution. Initially, the service invoker discovers the implementation code of the requested operation and its parameters by using the metadata that are produced before, and it constructs a dynamic method for real service invocation. For instance, “BookFlightService Param1 Param2” logical action is produced and “BookFlightService” has an implementation, namely “BookFlightServiceClass”, Param1 and Param2 have implementations, “Param1Class” and “Param2Class” respectively, and BookFlightService has an endpoint namely “BookFlightServiceEndpoint”. The parameters may have subtypes as mentioned before and they may have distinct implementations too, but for this example it is omitted for simplicity purposes. The mechanism that is described for this example can be applied recursively for subtype cases as well.

```
Class c = Class.forName(“BookFlightServiceClass”);
Constructor construct = c.getConstructor(new Class[] {URL.class, Service.class})
Object stb = construct.newInstance(new URL("BookFlightServiceURL”, null);
Class partypes[] = new Class[2];
partypes[0] = Class.forName(Param1);
partypes[1] = Class.forName(Param2);
Method meth = c.getMethod("BookFlightOperation", partypes);
```

Figure 5-6 Dynamic Method Generation Example

The java code in Figure 5-6 shows how a service call method is dynamically generated for the “BookFlightService” case. The information that is needed for dynamically generating a service call method is collected during preprocessing phase of the system and it is written to the metadata files as described before.

Actually, it is not sufficient to generate only a method for service invocation. The values of the method arguments are needed as well. The Logical/Physical map is

used for this purpose. The action handler makes sure that map contains the required syntactic values of logical parameters before the real service execution begins. After method construction, service invoker gets the real values of the logical parameters from Logical/Physical map and instantiate the dynamically constructed methods with the obtained values. The java code for the instantiation of the example above and the real service call is shown in Figure 5-7.

```
Object arglist[] = new Object[2];
 arglist[0] = “Param1Value”;
 arglist[1] = ”Param2Value”;
 meth.invoke(stb, arglist);
```

Figure 5-7 Dynamic Method Invocation Example

Param1Value and Param2Value are the real values of Param1 and Param2 logical objects respectively that are obtained from the Logical/Physical map. If the called service is an information providing service, it returns some values and handlings of these values are done with a similar mechanism that is applied for inputs with the help of reflection mechanism and java beans. If the returned type is a complex type, the sub-parts are obtained by using the properties of java beans. The generated code for complex types is in the form of java beans which provides a static interface for reaching the subcomponents of the returned values so the handling of them can be done automatically without a need for human assistance. The service invocation is automatically done by the software agent by using the mentioned mechanisms.

5.2 Logical/Physical Map

The logical/physical map is a very important structure for the proposed framework. It maps the logical objects that are used as action parameters by the planner and their syntactic counterparts and values of those syntactic counterparts which are required for real service calls. The graphical representation of the internal structure of the map is shown in Figure 5-8.

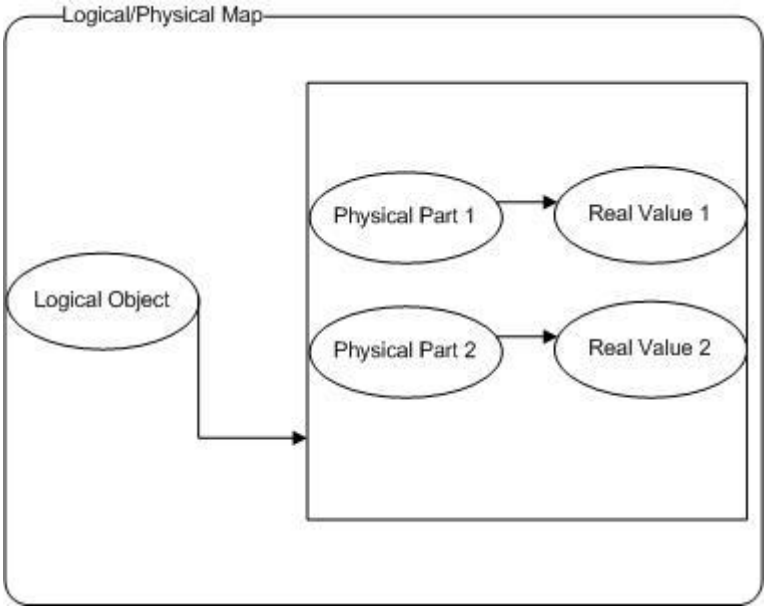

Figure 5-8 Logical/Physical Map

The Logical Object is the one that is used by the planner and it is defined in objects section of problem.pddl file. Physical parts are the WSDL counterparts of semantic types. Syntactic counterpart of a semantic type may be a complex type and it may contain subcomponents. Therefore more than one physical part may exist for a sematic type as shown in Figure 5-8. For the real service call, the values of the physical parts are needed and they are stored in real value parts.

Before each service call, the real values are obtained from either the user or other services; otherwise service invocation process does not happen. At the beginning of each session, that is, at the beginning of each user request, a fresh map is constructed. The syntactic counterpart of each PDDL object is constructed by associating the semantic and syntactic types as described earlier in this chapter. The constructed metadata files are used for this purpose and after physical parts are formed; their values are initialized to the value “unknown”.

When a logical action with logical object parameters comes to the execution module, the action handler examines the values of logical objects in the map. If they have real value components with value “unknown”, the information collector is

fired. The information collector asks the user for the real values of the physical components of logical objects if they are “unknown” in the map. If the user provides the values, the map is updated with these values and service invocation begins. If the user does not know the values asked, he/she tells the system that he/she does not know about that particular information through the provided user interfaces. In such a case, the unexpected event handler and planner collaboration tries to find a service that can provide that particular information. If such a service is found, the current plan changes and the information providing service is invoked with the same steps applied. The map is updated with the values that are collected as a result of calling the information providing service. Then, the action handler discovers that the required information is available in the map and allows service invocation to operate. As mentioned before, if the required information cannot be collected before the real service call, that is, if the map contains “unknown” values for the required logical object counterparts, the service invocation is not allowed .

A fresh map is initialized at the beginning of each session. The construction of the map is dynamic and it is done through out the current session according to the logical action definition. There may be several logical actions that are grounded with some logical objects but the syntactic structure of distinct services may be different. For instance consider the services “BookFlightService ?Flight – Flight ?Person – Person” and “BookMedicalFlightService ?Flight – Flight ?Person – Person” and pddl objects “Flight1 – Flight and Person1 – Person”. The two services given above are two distinct services, but their semantic parameter types are the same. After the planner performs grounding, two distinct logical actions are generated with the same logical object parameters as follows:

BookFlightService Flight1 Person1

BookMedicalFlightService Flight1 Person1

The intended semantic meaning of “Flight1” and “Person1” objects is the same in two grounded services, since “Flight1” and “Person1” are used with the same meaning in mind by the user. However, some syntactic differences may occur for

the physical definition of logical objects. For instance “Flight” type may have a WSDL counterpart such as “xsd: int” for “BookFlightService”, but a complex type counterpart that contains both a string and integer values “sequence (xsd: int, xsd: string)” for “BookMedicalService”. In this work, this problem is solved to some extend by destroying the map after each service call. When “BookFlightService” is invoked or it is discovered that it cannot be invoked because of some unavailable information or service unavailability, the physical counterparts of logical objects “Flight1 and Person1” are destroyed. When “BookMedicalFlightService” is called, physical counterparts of Flight1 and Person1 objects are constructed again according to the definitions found in the WSDL of “BookMedicalFlightService”. If destruction is not conducted after each action call, syntactic incompatibilities may arise.

The proposed destruction mechanism is not sufficient especially when the required information is gathered not from the user but from another service. For instance service “A” requires information which can be produced by service “B”. What “B” provides to “A” is desired in semantic level but not in syntactic level. For the above case, “BookFlightService” requires the information about the object “Flight1” which has a semantic type “Flight” and “ProposeFlightService” produces the value of “Flight1” object which has a semantic type “Flight”. However there may be syntactic differences between the requested and the provided “Flight1” object. This problem is not so difficult, since the agent just needs to make some syntactic analysis before placing the values to the map. However in this work, it is assumed that, if particular semantic information is requested by another service, the provided information is syntactically compatible for simplicity purposes. This problem will be solved in future extensions of this work.

5.3 Action Caching Mechanism

Planning problems are generally very complex and time consuming problems to solve. Although the use of Simplanner provides an important amount of advantage with respect to time, precompiled solutions are still very valuable. The planner requires some deliberation time before proposing an action and it may also give

wrong decisions for the initial solution and needs some more steps for recovering the wrong decisions that is given because of insufficient time.

Since the number of distinct user requests from the service composer agent can be infinitely many, it is impossible to hold the answer of each request in advance; but this does not mean that previously found solutions are not usable. Users may have similar requests from the system, so if previously found solutions are saved, they can be used directly later.

In this work, such a previous experience caching mechanism is proposed. After each successful session, the pddl action definitions are updated with some supporting metadata. During the processing of each session, the software agent keeps track of successfully executed services and the initial and final logical states of the current session. After the successful termination of the session, the software agent constructs a new pddl action with a precondition which represents the initial state of the current session and with an effect which represents the final state of the current session. Parameters of the action are determined according to the requirements of the constructed precondition and effect parts. After constructing the new pddl action, the agent compares the available pddl actions with the one that is currently constructed. If there does not exist any pddl action with the same precondition and effect, the new pddl action is added to the pddl action definitions in domain pddl file. After the addition of the newly constructed action to the domain.pddl, the metadata of the new action is written to an xml file namely "complexActions.xml". The metadata contains the successfully executed logical actions in an order which represents the components of the constructed complex action. The structure of xml file that holds the components of the complex action is shown in Figure 5-9.

```
<complexActions>
<action name=”LogicalActionName”>
<subAction name= “SubAction1”>
<subAction name=“SubAction2” >
……..
</action>
…….
</complexActions>
```

Figure 5-9 Complex Action’s XML Structure

Simplanner usually proposes the shortest path to the solution. If a new problem is posed to the system that is similar to the ones that were already solved by the system before, Simplanner will most probably propose the use of the already constructed complex action which achieves the goal in much less number of steps. Less number of steps is valid only for logical actions; physically, the same number of steps is required for performing the goal.

At the beginning of each session, an in memory data structure is formed that contains the information about complex actions by parsing the “complexActions.xml” file. If the planner proposes a complex action to the execution module, the action handler component of the execution module examines the action and decides if it is a complex action or a usual action by using the information that is available in the data structure constructed for complex actions.

If a complex action is proposed by the planner, the action handler detects it and it performs the grounding of subactions by using the parameters of the complex action and the other automatically generated logical objects. The execution handler does not request any other action from the planner until the termination of the real execution of subactions. During real execution of subactions, unexpected problems may arise as in usual actions such as service and information unavailability. In that case, the same unexpected event handling procedure is applied with some small modifications. If the service that corresponds to a subaction is unresponsive, it is invalidated by the unexpected event handler. The planner does not consider invalidated services for other decisions. In complex action case, a similar approach

is used. Instead of just invalidating the subaction that is unresponsive, the complex action itself is invalidated too by the unexpected event handler. The transactional service execution strategy is preserved for complex actions too but some more processing is needed for complex actions that are described in Chapter 6 where transactional issues for WSC are discussed.

In order to clarify the mentioned approach, consider the following example. Suppose the user requests a goal "<isBookedFor Flight1 Person1>" with the logical objects "Flight1 – Flight and Person1 – Person" and the initial state is empty. The software agent executes "CreateFlightAccount", "ProposeFlight" and "BookFlight" services respectively with required parameters to solve the problem. If the request is successfully handled, an action definition is added to the domain pddl definition, and the components of the complex action are added to the "complexActions.xml" file as in Figure 5-10:

```
(:action newAction
:parameters (?Flight – Flight  ?Person - Person)
:precondition (and (valid newAction)
(agentHasKnowledgeAbout ?Flight)
(agentHasKnowledgeAbout ?Person)
)
:effect (and  (isBookedFor ?Flight ?Action))
)
```

```
<complexActions>
<action name = "newAction">
<subaction name="CreateFlightAccount"/>
<subaction name="ProposeFlight"/>
<subaction name="BookFlight"/>
</action>
</complexActions>
```

Figure 5-10 Example Complex Action

The planner proposes "complexAction" as a logical action when a problem similar to the one solved previously is given to the system. In the example above, when the planner proposes "newAction FlightObj PersonObj" to the execution module, the action handler will understand that it is a complex action and it will get the

subcomponents of “newAction” which are “CreateFlightAccount”, “ProposeFlight” and “BookFlight”. The action handler does the grounding of these actions by examining the parameter types and available logical objects. If an object is needed with type “Flight” or “Person”, the parameters of “newAction”, namely “FlightObj” and “PersonObj” will be used. If a parameter is needed with some other type, logical objects that are automatically generated at the preprocessing phase are used for grounding. After grounding of the subactions, the operation that is applied to the usual actions is applied with no distinction except for the transactional issues discussed in Chapter 6. The executor does not request any other new action from the planner until the completion of real execution of subactions, as mentioned before.

CHAPTER 6

TRANSACTION ISSUES FOR WEB SERVICE COMPOSITION

One of the most important goals about the automated web service composition that this work focuses is to deal with nondeterminism of services and partial observability of the environment. Simplanner that is chosen for logical action construction for WSC is very effective for dealing with these problems. The service and information unavailability is effectively handled by the planner. In case of such problems, the planner tries to find alternative paths for solving the problem. However, sometimes it is not possible to discover alternative ways when unexpected situations arise, so the session should be terminated unsuccessfully.

The planner handles unexpected situations at the logical level which is not sufficient in fact. Some physical mechanisms are also needed for handling such cases. The proposed system works step by step and executes actions physically. If any problem occurs in later steps of processing and the previously executed actions have some world altering effects, some undesired situations may arise.Transactional execution is the solution that we propose to solve the mentioned problem. Transaction is a concept generally related with databases. It represents the logical unit of work for database management systems. Although transaction concept is very tightly coupled with databases, it is widely used in distributed systems for the same purposes in an application level.

Some specifications, such as WS-Coordination, WS-Atomic Transaction and WS-Business Activity, are proposed to enable transactional features during web service collaboration ([7], [12], [13]). In this work such mechanisms are used for recovering undesired effects. In section 6.1 an introduction to the WS-Transaction frameworks is presented. In section 6.2, the integration of these frameworks to the presented system is discussed.

6.1 WS-Transaction Frameworks

The aim of using transactions is to provide ACID properties (i.e. atomicity, consistency, isolation and durability) to the execution of a composed web service. ACID properties are defined as follows [45]:

- Atomicity: If the result is successful, then all operations happen, and if it is unsuccessful, then none of the operations happen.
- Consistency: The application performs valid state transitions at completion.
- Isolation: The effects of the operations are not shared outside the transaction until it completes successfully.
- Durability: Once a transaction successfully completes, the changes survive failure.

In order to achieve ACID properties for applications that involve multiple web services collaborating with each other, some specifications are proposed. These specifications are WS-Coordination [12], WS-Atomic Transaction [13] and WS-Business Activity [7]. WS-Coordination specification is the base of the other two specifications.

WS-Atomic Transaction is generally used for business activities that do not require too much time to be completed. WS-Atomic transaction specification provides all ACID properties by applying two-phase commit protocol. That is, all sub-operations of the transaction will be entirely conducted successfully or none of the sub-operations will be performed. The intermediate results of the whole operation are not visible by the users.

WS-Business Activity specification [7] is more flexible but does not provide all ACID features. This specification is designed for long running business activities. In this case, the intermediate results are visible to the users (isolation property is not provided) and effects of the sub-operations are conducted before the commit command. Therefore, instead of using usual rollback operation, some compensation

techniques are needed. During compensation some problems may arise which may disturb atomicity. In such cases human intervention is needed.

WS-Business Activity specification does not provide a complete transactional behavior but on the other hand it brings an important advantage. In two-phase commit protocol which should be used for complete transactional behavior in distributed systems, some locking mechanisms are used. As a result, the same resources cannot be used by other users until the business activity ends. If business activity takes long time and the number of system users is high, the system cannot be responsive which is not acceptable.

As mentioned before both of the specifications that are described above are based on WS-Coordination specification. WS-Coordination specification has three components which are activation service, registration service and coordination service. Figure 6-1, adapted from [45], shows the structure of WS-Coordination framework graphically.

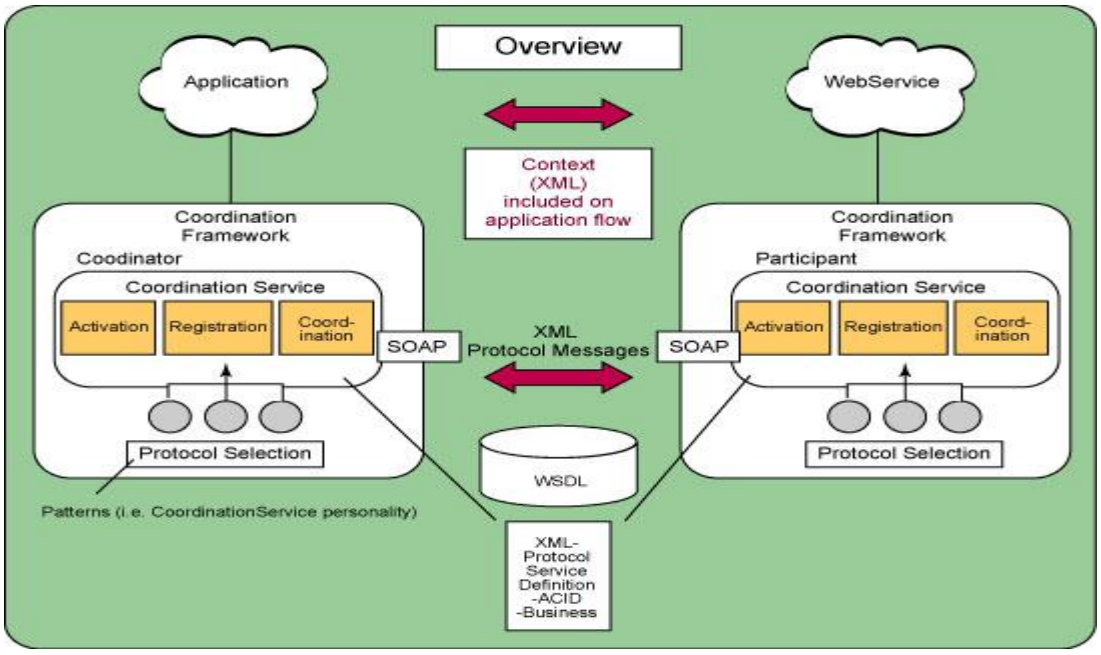

Figure 6-1 WS-Business Activity Framework

The main application and participating web services communicate through the coordinator. The same architecture is used for both WS-Atomic transaction and WS-Business Activity with some differences in lower level details. According to [45], the functionality of WS-Coordinator can be stated as follows:

Activation Service:

The main use of activation service is to start a fresh business activity.

Registration Service:

The registration service enables distinct web services that are participants of the current business activity to be enrolled to the activity.

Coordination Service:

The coordination service controls the operations of participating services, the current states of them and gives commands to the services for final decision of the conducted transaction using the selected coordination protocol.

The details of the mechanisms and some sample scenarios can be found in [45] and [46].

The WS-Business Activity framework specification which is used in this thesis has a shortcoming. The specification does not include the protocol between the application and the coordinator. It only defines the protocol between the coordinator and the web service. However, in [47], a protocol between the main application and the coordinator is proposed namely "Web Services – Business Activity Initiator Protocol" which is an extension to the previously described WS-Business Activity protocol. This extension is used in this thesis.

As mentioned in [47], WS-Business Activity has two types of coordination mechanisms: BusinessAgreementWithParticipantCompletion protocol and BusinessAggrementWithCoordinatorCompletion protocol. In the first protocol, the participating web service performs its activity immediately after a service request comes from the main application and informs the coordinator when it finishes its job either successfully or unsuccessfully. In the latter case, the web service does not

complete its job but waits for a command from the coordinator for completion. The remaining mechanisms are same for both protocols. The decision about which coordination protocol is to be used depends on the choice of the service provider.

Figure 6-2 adapted from [47] shows the possible states of participants during the execution of the transaction and possible messages between the coordinator and participants.

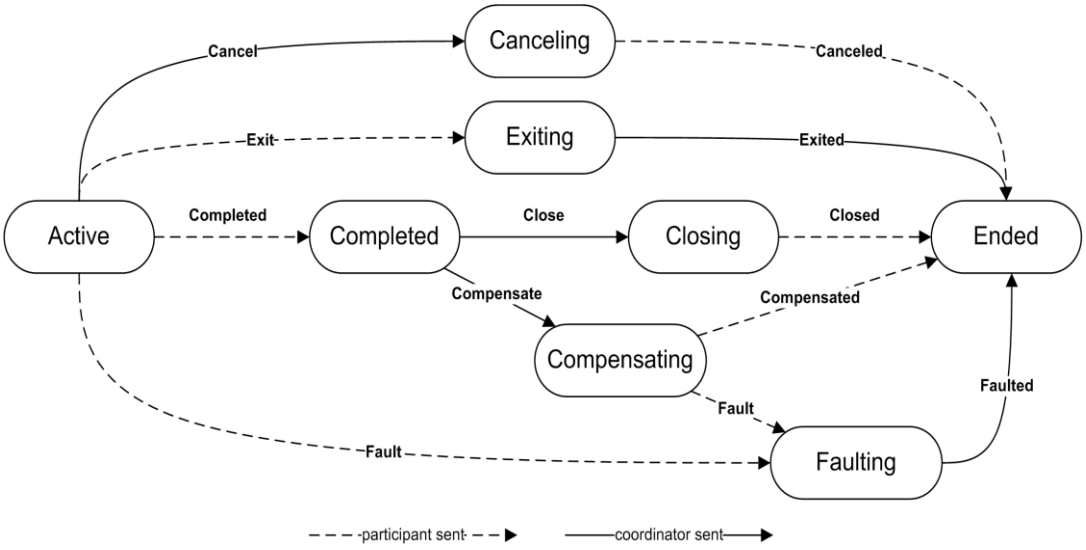

Figure 6-2 WS-Business Activity State Diagram

The states shown in the figure are defined for the BusinessAgreementWithParticipantCompletion coordination protocol. The states for the other protocol are similar and can be found in [47].

When a request comes to a participating web service, it starts its work, and it is in active state during that period. If it finishes its work successfully, it sends the command “Completed” to the coordinator and the coordinator changes the state of the participant to “Completed”. If the service could not do its job because of some unexpected failure it sends “Fault” message to the coordinator, and the coordinator

changes the status of the participant to “Faulting” state. After the coordinator gets the decision commands from the business initiator, it sends commit or rollback commands to the participants. If a commit decision is made, “Close” signal is sent to all participants by the coordinator. If a rollback decision is made, “Compensate” signal is sent to all participants by the coordinator and the required state changes are conducted as in Figure 6-2. If a problem occurs during compensation of one of the services, atomicity is destroyed and human intervention is needed. Some other commands exist between the coordinator and participants such as exit and cancel. The details of them can be found in [47].

The messages that are sent by the coordinator to the participants such as “Close”, “Compensate” are sometimes needed to be dictated by the main application. “Web Services – Business Activity Initiator Protocol” proposes some other commands for such cases. Business activities may contain sub-transactions which are atomic themselves but they are independent from other sub-transactions. In order to handle such cases, WS-Business Activity framework allows mixed outcomes, in which independent participants may have independent target states. For instance, suppose a sub-transaction compensates but other participants commits. If mixed outcome is not required, the atomic outcome is used, that is the target state for all participants is the same; participants all commit or all compensate. In “Web Services – Business Activity Initiator Protocol”, there exists commands for handling both mixed outcomes and atomic outcomes. In this work, the use of atomic outcome is sufficient, so only commands of atomic outcome are required in this work. The commands that are provided by Web Services – Business Activity Initiator for atomic outcome are as follows [47]:

listParticipants: As a result of firing this command, the coordinator provides the status of each participant to the main application.

closeAllParticipants: As a result of firing this command, the main application dictates that the transaction should be committed. After the coordinator receives this command, it sends “Complete” command to each participant.

cancelOrCompensateAllParticipants: As a result of firing this command, the main application dictates that the transaction should be aborted. After the coordinator receives this command, it sends “Cancel” or “Compensate”command to each participant according to their status. If participant is active, “Cancel” command is sent, otherwise “Compensate” command is sent. As mentioned before if any problem occurs during compensation, human intervention is needed to ensure atomicity.

The details of the mentioned commands and information about some other utility commands can be found in [47].

6.2 Integration of WS-Transaction Frameworks to the Proposed System

The unexpected event handler of the proposed system uses two mechanisms for handling the undesired situations. The recovery of the logical state is done by the help of planner as described in Chapter 3 and Chapter 4 and the physical recovery is done by the help of WS-Business Activity framework.

Among the available WS-Transaction frameworks, WS-Business Activity framework is more suitable for the proposed system because long running sessions are usually conducted in the system. Generally a high amount of user interaction is needed for information collection and planner needs some deliberation time before the real service execution all of which require time. If WS-Atomic Transaction framework is adapted to the system, system cannot respond to other users’s requests since resources are locked during each session which prevents the calling of the same services concurrently.

WS-Business Activity framework is used for coordinator-participant communication and Web Services – Business Activity Initiator Protocol is used for the communication between the service composer agent and the coordinator in the proposed system. The implementations of both protocols are done by Apache Kandula project [44] and Kandula is adapted to the proposed system.

The coordinator implementation of the Kandula [44] project is deployed as a web service and the world altering service calls are done through the coordinator. When the service composer agent decides to make a real service call, it checks to see if the service is a world altering or information gathering service by examining the logical effects of the logical action that corresponds to the service to be executed. If it is an information gathering service, the procedure that is described in Chapter 5 is used. If it is a world altering service, the service composer agent invokes that service in transactional activity.

World altering services are assumed to implement WS-Business Activity participant specification in this work. This assumption is not needed in fact, because it is possible to find out if the web services implement that specification or not from their WSDL definitions. However, such an assumption is still made for simplicity purposes. If such an assumption is not made, WSDL definition of each world altering service should be examined and if they do not satisfy the needs of the specification, they must be removed from the search space of the planner.

The coordinator that is deployed as a web service keeps status of all world altering web service participants through interfaces provided by the WS-Business Activity. However, all commands that are sent to the participants by the coordinator are determined by the service composer agent itself. The service composer agent sends the commands to the coordinator through Web Services – Business Activity Initiator protocol according to its decisions. The coordinator works as a proxy that conveys commands from the service composer agent to the web services.

The service composer agent constructs a coordination context at the beginning of each session, and it starts a new business activity by conducting the required communication with the coordinator that is deployed as web service beforehand. The service composer agent starts a business activity with an atomic outcome. The mixed outcome that allows nested transactions is not required in this system. The mixed outcome choice may be helpful for complexActions case described in Chapter 5, but another strategy is used for handling transactional issues of complexActions, which is described later. After a business activity is started by the

agent, a coordination context is requested by the coordinator before a call is made for each world altering service. This context is sent to the web service with other service arguments. After the message is delivered to the web service, it registers itself to the current activity by communicating with the coordinator and then starts its work. After the completion of its work, it informs the coordinator about its status and sends the outputs of the request to the service composer agent if it can.

As mentioned before, WSDL definitions of world altering web services include the transactional features as well. For instance, a booking flight service with inputs “user account”, “flight number” and “personname” has input message part as follows in its WSDL definition.

```
<message name="BookFlightInputMsg">
  <part name="transactionalContext" type="tns:contextChoiceType" />
  <part name="useraccount" element="tns:UserAccount" />
  <part name="flightnumber" type="xsd:int" />
  <part name="personname" type="xsd:token" />
</message>
```

Figure 6-3 WSDL Message with Transaction Parameter

The type “contextChoiceType” is defined according to the “WS-Business Activity” specification.

When transaction is required, the dynamic service call mechanism that is described in Chapter 5 changes slightly. The coordination context that is requested by the service composer agent from the coordinator is transported to the web service. For the example above, the first input argument represents the context and it is sent to

the real service by the dynamically constructed method. The additional required data can be stated for the example above as in Figure 6-4.

```
Constructor transconstruct = transcls.getConstructor(new Class[]{CoordinationContext.class});
arglist[0] = transconstruct.newInstance(ctx);
partypes[0] = ContextChoiceType.class;
```

Figure 6-4 Dynamic Transactional Method Construction

"arglist[0]" is the context that is returned from the coordinator for a particular web service and "partypes[0]" is the implementation code of custom type definition. "partypes[0]" along with the other input argument types is used for constructing the method and "arglist[0]" is used along with the other input values for calling the dynamically generated method. The mechanism used is described in Chapter 5 in detail.

When the software agent decides that the current session can be terminated successfully, it sends the command "closeAllParticipants" to the coordinator which informs each participant about the situation. The execution component of the agent controls if there exists a remaining goal to be achieved before requesting a new action from the planner. If there does not exist any remaining goals, that is all requests are handled, the software agent decides to terminate the session successfully and informs the world altering services about the decision by means of issuing the command "closeAllParticipants".

When the software agent decides that the current session should be aborted, it sends the command "cancelorCompesnateAllParticipants". The planner tries to find some alternative paths when an unexpected situation arises. It is possible that such paths do not exist from time to time. In such cases a solution cannot be found for the requested goal and sometimes even if unexpected events do not happen, it is

understood that planner cannot provide a solution to the problem after some steps. In such cases, the software agent issues the command "cancelorCompensateAllParticipants" to the coordinator for the current activity and the coordinator informs the web services for executing their compensation operations.

The problem of demonstrating the inexistency of a plan is very difficult for not only Simplanner but all existing planners. Generally the whole search space should be examined which is impossible for big environments such as WSC domain since a huge amount of time is needed. Some admissible heuristics are needed for understanding the plan inexistency. In WSC case, it is very rare that the same service is called with the same input values in a single session. Although sometimes a need for such calls occurs, such rare cases are not considered for timely response of all other cases. In this system abort decision is made if the same service call is proposed by the planner with the same logical object parameters and with the same physical values (obtained from the physical/logical map). Some more restricted admissible heuristics can be applied such as considering the current logical state. That is, abort decision is made if the same service call is proposed by the planner with the same logical object parameters and with the same physical values in the same current state. In fact all these session abort heuristics are domain dependent and they cannot be used in domain independent manner.

As described in Chapter 5, action caching mechanism is implemented in the proposed system. In such cases a logical action is composed of multiple physical actions. The service calls of the physical action components are done by conforming the transactional rules mentioned before, that is, world altering service component calls are done through the coordinator. Suppose a complex logical action is proposed by the planner where one of the physical service calls that is a part of that complexAction failed. In such a case, the logical complexAction and logical actions that correspond to the faulted physical service are excluded from the logical state and some other paths are tried to be discovered by the planner. If some other path is found and the problem is solved successfully without using the previously used complexAction some problems occur. If a new path is found that enables the

solution of the problem, successful termination message is sent by the service composer agent to the coordinator, and compensation actions are not called for the executed physical action components of the failed complexAction.

In order to prevent such undesired situations, the effects of complexActions are made visible after not all physical action components are called but each physical is called. The logical effects of each physical service component are made visible after each service call. Therefore, the planner finds alternative paths in case of a complexAction failure by considering the logical effects of successfully called physical components of the complexAction. As a result, when successful termination decision is made by the software agent, the compensation operations of physical components of failed complexAction are not needed to be called since the planner considers their logical effects during discovering the new path. If alternative path could not be found, usual compensation mechanism is applied to all the previously executed world altering services.

CHAPTER 7

CASE STUDY: TRAVEL DOMAIN

In this chapter, a simple case study is presented in order to illustrate the implementation and functionality of the proposed system. The system is highly resistant to unexpected real world situations and it provides timely response. The current system is applicable in relatively small environments. It is applicable in some particular domains or some prefiltered environments. Although the proposed system provides very valuable features for real world web service composition scenarios, it is not scalable. Therefore it cannot be used in an environment where millions of web services exist.

The scalability problem occurs because of the used AI planner. Not only the used AI planner (Simplanner) but all domain independent AI planners have the same problem. They cannot work with big domains where more than a few thousands of actions exist. Some domain knowledge is needed to make the planner scalable, but in WSC domain such domain knowledge is very limited. The scalability might be provided not in the planner level but in preprocessing level. According to the user's request, some smart elimination on available services can be done for reducing the search space [35]. Such a prefiltering mechanism can be added to this work in future extensions. Here, the case study is conducted with the assumption that prefiltering has been done beforehand.

This case study is based on a travel domain which is used as a data set in [48]. (The used travel ontology is provided in Appendix). The web services presented in Figure 3-1 are used together with some additional web services.

Additional web services are “BookMedicalFlight”, “CreateMedicalFlightAccount” and “RequestMedicalFlight”. These three services provide the same functionality as “BookFlight”, “CretaeFlight” and “RequestFlight” services given in Figure 3-1. They only have some syntactical differences. This case study contains two parts that focus on information and service unavailability respectively.

7.1 Case 1: Information Unavailability

In this scenario, the user requests to reserve a vehicle transportation using the constructs of the available travel ontology (see Figure 7-1).

```
<Patient rdf:ID="Patient_0"/>

<VehicleTransport rdf:ID="TransportToHospital">

   <isBookedFor rdf:resource="#Patient_0"/>

</VehicleTransport>
```

Figure 7-1 Example User Request

This request is a high level request that is constructed using the travel ontology constructs (some OWL individuals and their relationships). This request, along with the constructs generated by the software agent, is converted to PDDL and the planner starts to work on the problem. The service “RegisterPersonWithTransport” does the booking operation, but it has a precondition that it requires a valid customer account and some other inputs. The service “CreateVehicleTransport” satisfies the precondition of the service “RegisterPersonWithTransport”. As a result the planner produces a plan given in Figure 7-2.

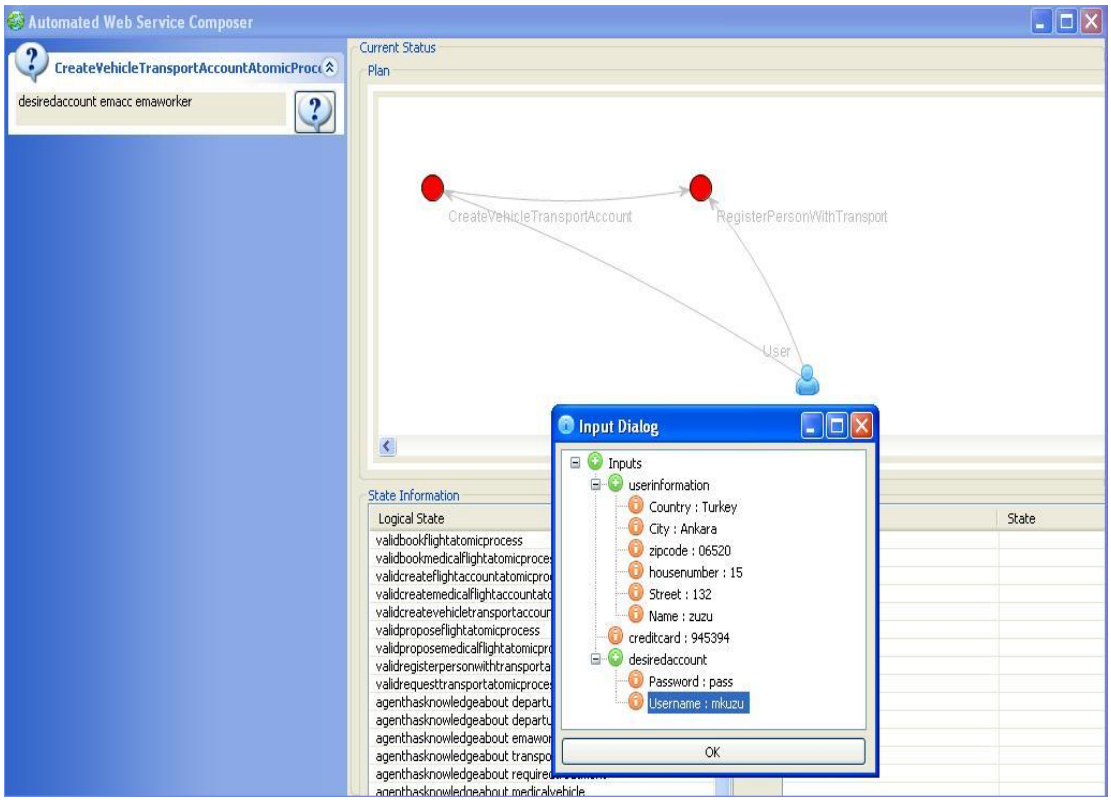

Figure 7-2 Initial Plan Generation

The planner proposes an action with some logical parameters such as “desiredaccount”, “emacc” and “emaworker” for this particular case. Before service composition begins, the logical/physical map is constructed and the physical counterparts of all logical objects are written to the map by using the OWL-S grounding part and WSDL definitions of services. “Input Dialog” is prompted by the software agent to the user in order to get the physical values of logical objects. The user inputs the required information and the map is updated with the given values. By using the provided inputs and previously constructed service client stubs, the software agent does the real service call.

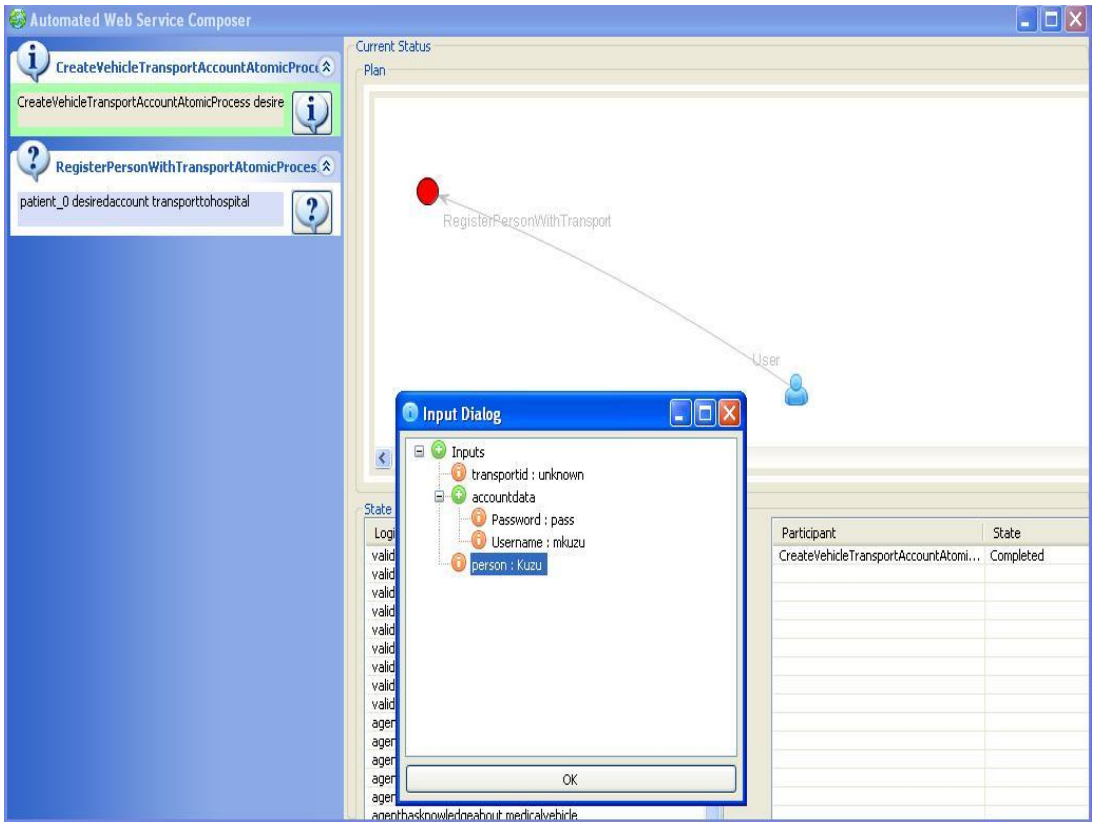

Figure 7-3 Unknow Information Scenerio

Since, “CreateVehicleTransport” service has world altering effects, its call is done through the WS-Business Activity coordinator that is constructed at the beginning of the session. The service “CreateVehicleTransport” implements the “BusinessAgreementWithParticipantCompletion” protocol that is described in Chapter 6 and the coordinator changes its state to “Completed”. After calling “CreateVehicleTransport”, the precondition of “RegisterPersonWithTransport” is satisfied and the agent prompts the “Input Dialog” to the user that asks physical counterparts of the logical parameters. The user does not know the “transportid” so that input cannot be provided. “Transportid” field is the physical counterpart of “transporttohospital” logical object. Since its real value is left “unknown”, the software agent removes the “agentHasKnowledgeAbout transporttohospital” logical statement. “RegisterPersonWithTransport” requires that “agentHasKnowledgeAbout transporttohospital” is true, so the planner searches for

a service that provides that effect. In other words, the planner searches for an action that provides the input that is required but the user does not know.

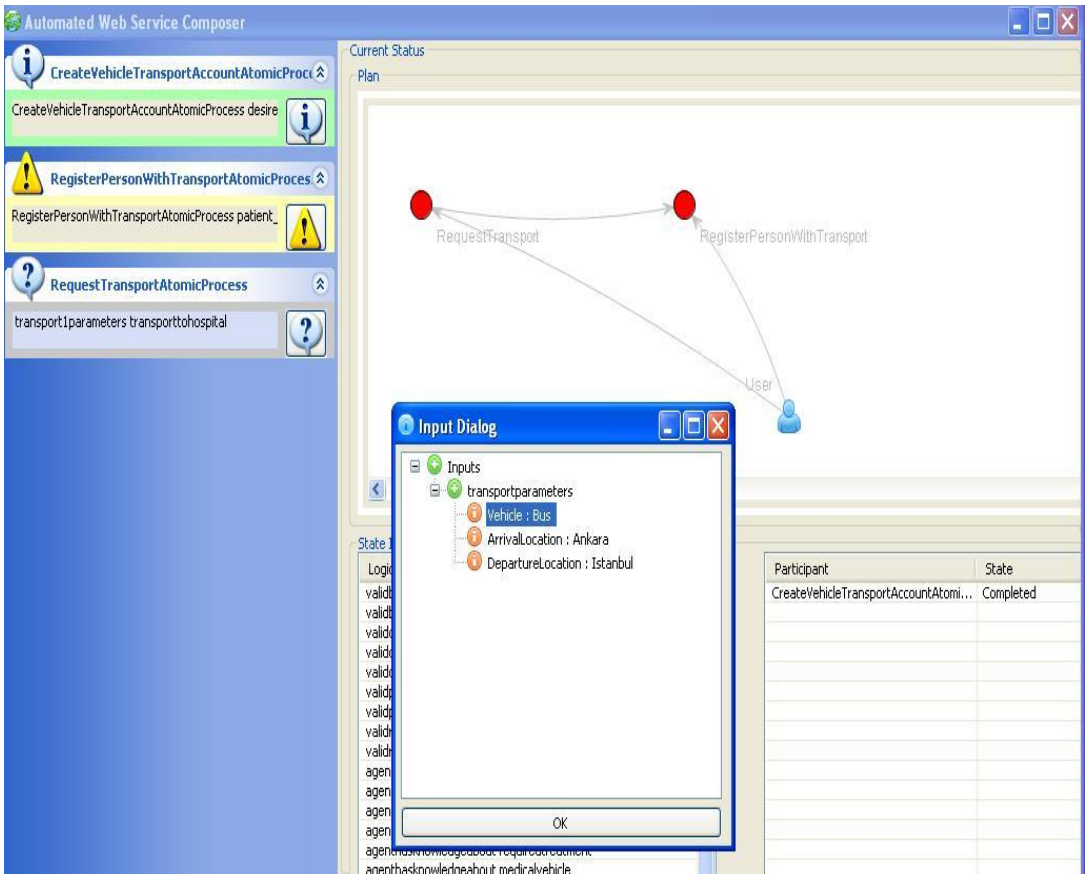

Figure 7-4 Information Providing Service Scenario

Since the required inputs for “RegisterPersonWithTransport” service could not be provided by the user, it is not called. Some other service “RequestTransport” service is able to provide the missing information and therefore the planner discovers that service. The required inputs are requested from the user for firing the “RequestTransport” service and after the required information is collected, the real service call is done.

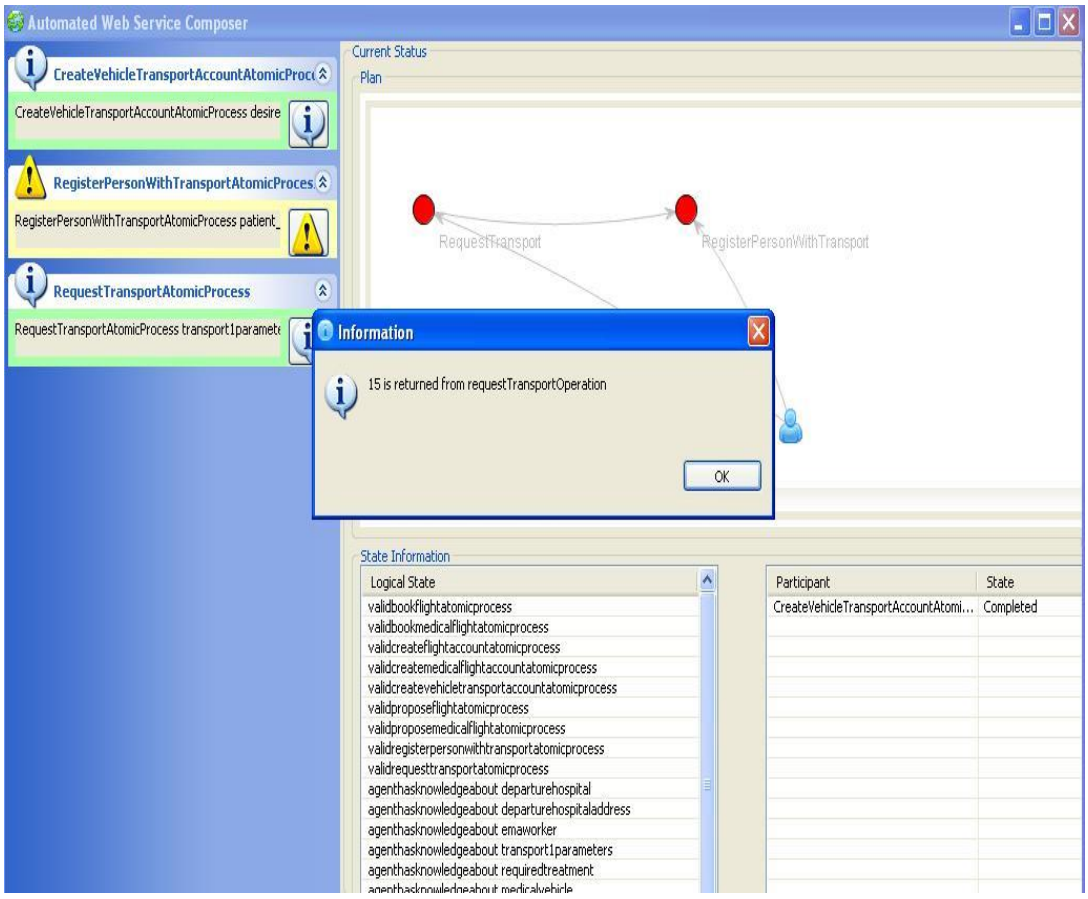

Figure 7-5 Physical/Logical Map Update Scenario

The service “RequestTransport” is called with the user provided input arguments and the returned value is displayed to the user. The returned value is also put in the logical/physical map. In some rare cases, human intervention is needed since dynamically constructed objects do not have versioning as the ones provided initially. The dynamically constructed objects are destroyed after each service call, but in some special cases the same dynamic object can be requested by more than one action prior to its consumption. In such cases, users will change the values according to the returned values. Since “RequestTransport” service does not have a world altering affect but only provides information, it is called directly, not in collaboration with the business activity coordinator.

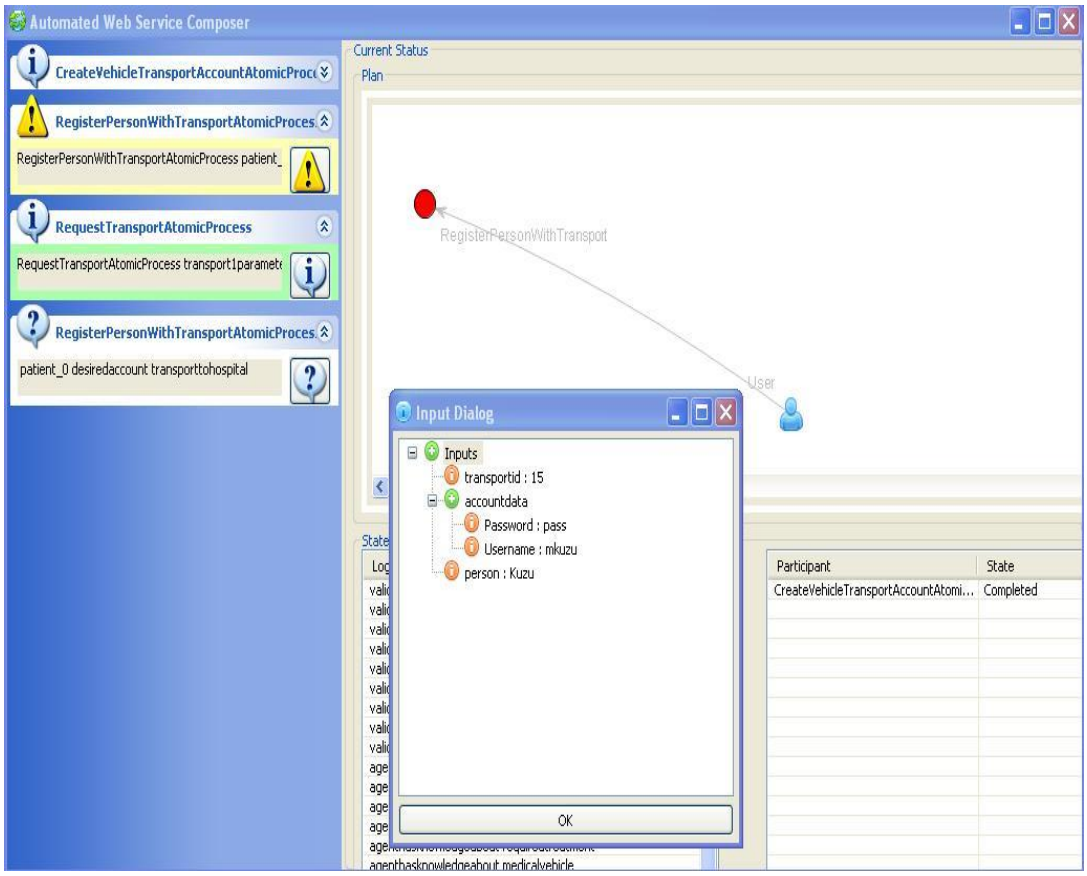

Figure 7-6 User and Service Provided Inputs Scenario

Since the "transportid" is provided by another service, the required inputs for "RegisterPersonWithTransport" service become ready to be executed and real service call is done by the software agent by using these inputs.

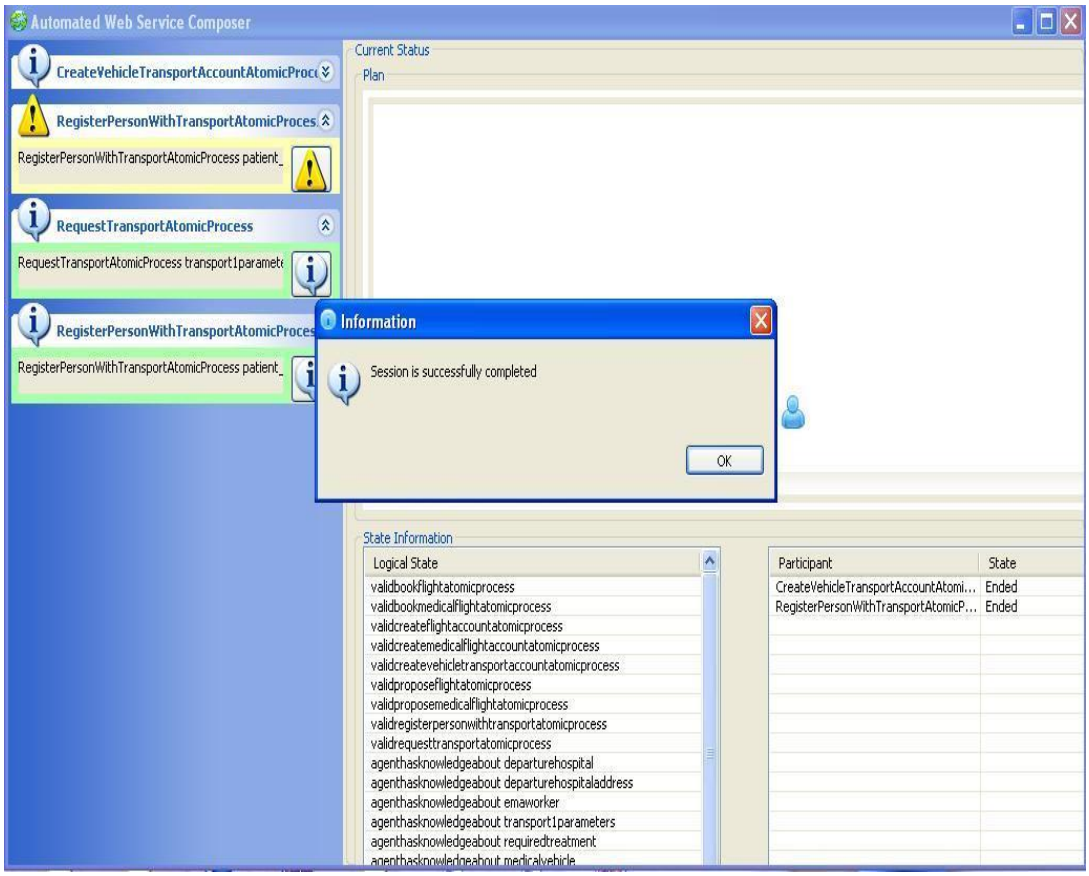

Figure 7-7 Successful Termination Scenario

After the service "RegisterPersonWithTransport" is executed, the session is terminated successfully since the user's goal is reached. "RegisterPersonWithTransport" service has a world altering effect. Therefore, its call is done through the business activity coordinator. After a successful session, the software agent sends "closeAll" signal to the coordinator which then sends it to each participant. Participants change their state to "Closed" and "Ended" respectively according to the WS-Business Activity specification.

7.2 Case 2: Service Unavailibality

The presented scenario in Case 1 is a simple scenario that demonstrates the working mechanism of the proposed system. However it does not illustrate all possible

situations that might be encountered. Some other important situations are demonstrated as another case. In this case, the user tries to reserve a flight and some unexpected situations occur (i.e. service failures). Only the most important parts of this case are presented in order not to repeat the similar things.

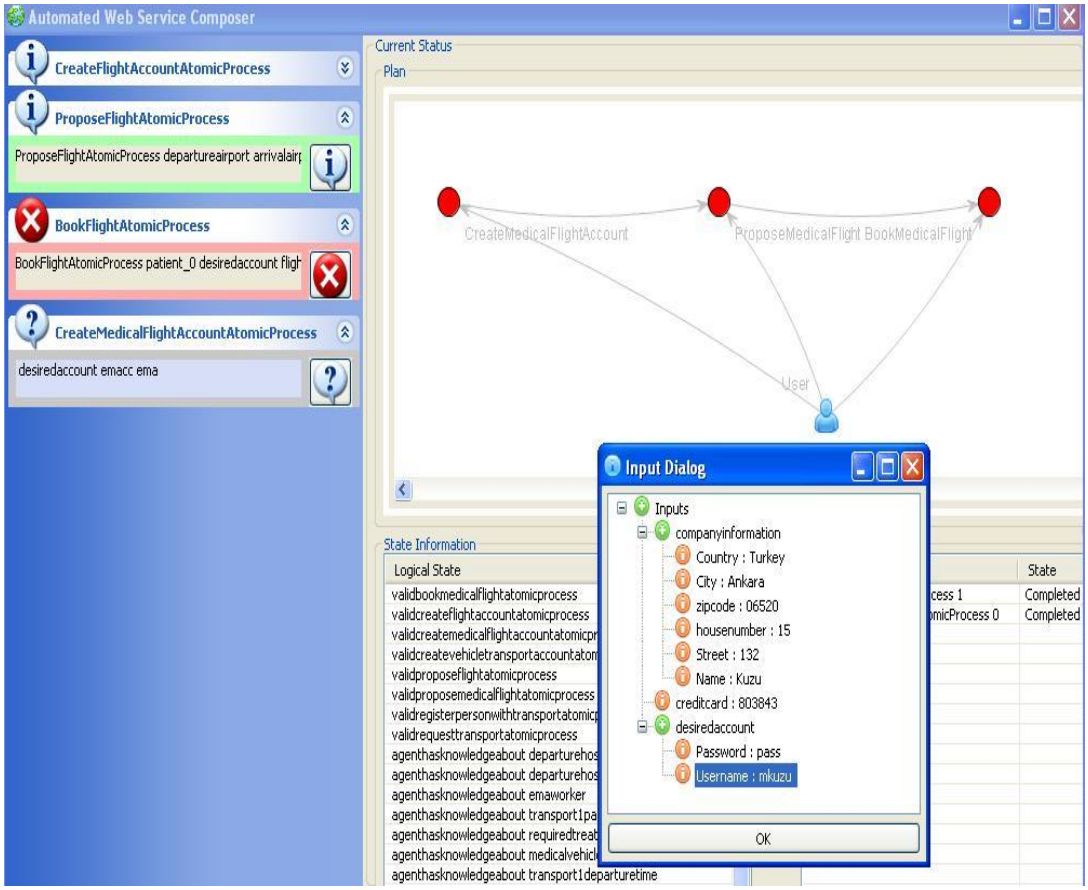

Figure 7-8 Service Failure Scenerio

Initially the software agent discovers a plan which contains services “CreateFlight”, “ProposeFlight” and “BookFlight”. The services “CreateFlight” and “ProposeFlight” are successfully executed, but during the execution of “BookFlight” service, the execution fails because of some network problems. The software agent then removes “validbookflightatomicprocess” logical statement form the current state and tries to

find some alternative paths in order to respond the user request. Since "validbookflighatomicprocess" statement is a precondition for firing "BookFlight" service and it is not available any more, that service becomes unavailable for the planner. The planner discovers a new path for achieving the goal by doing some dynamic replanning. The new plan contains "CreateMedicalFlight", "ProposeMedicalFlight" and BookMedicalFlight" respectively.

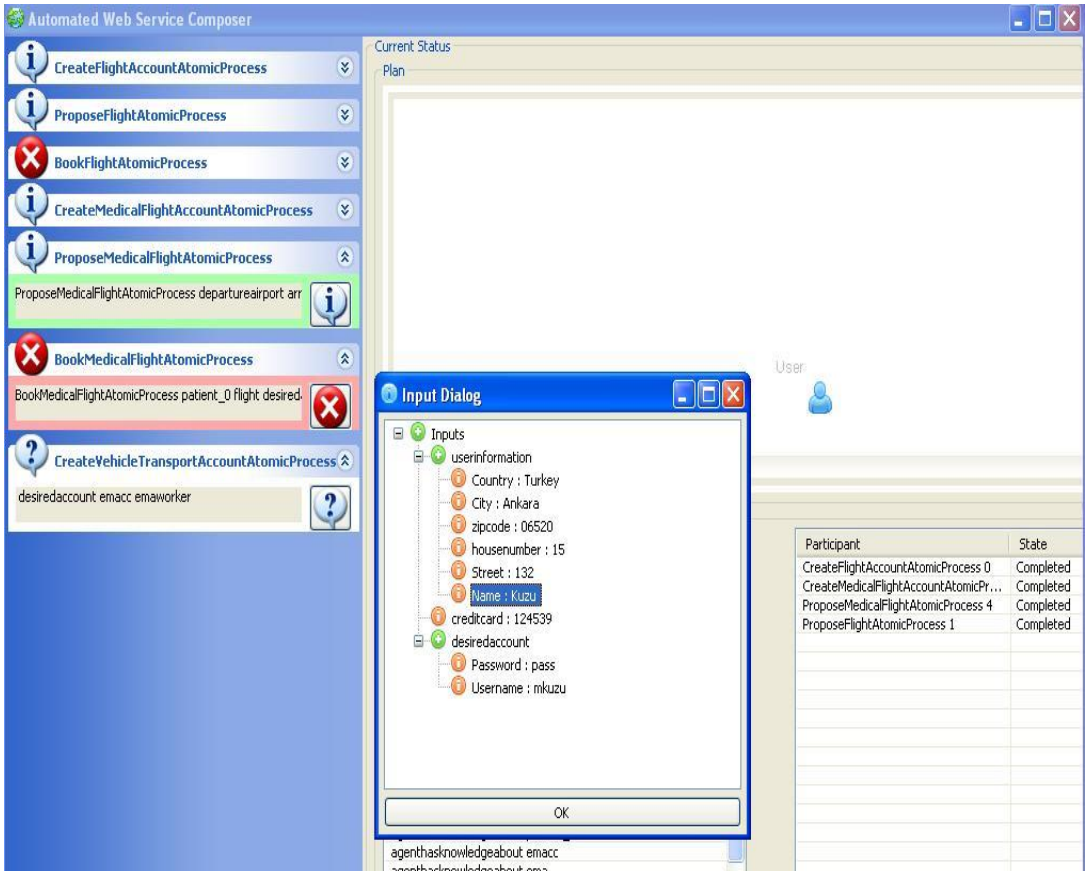

Figure 7-9 Unsolvable Problem Scenario

The services "CreateMedicalFlight" and "ProposeMedicalFlight" are executed successfully. During the execution of "BookMedicalFlight" service, a failure occurs because of service unavailability. The software agent removes "validbookmedicalflightatomicprocess" from the current state so that the service

will not be considered as an available action any more for this particular session. This time planner cannot produce an initial plan but proposes the best available action that is “CreateVehicleTransport” service.

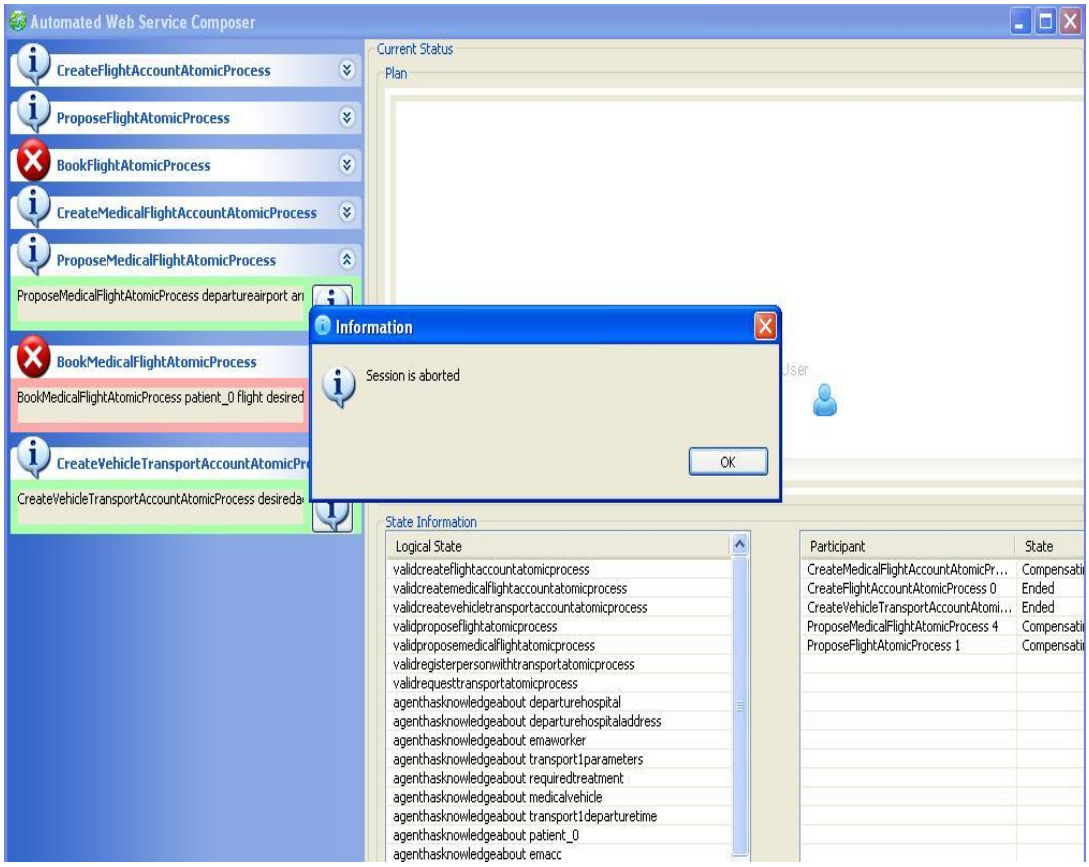

Figure 7-10 Session Abort Scenario

After the service “BookMedicalFlight” is failed, the planner cannot find a new plan and starts to search the state space. During that state space search, the same action, with the same logical parameters, is proposed by the planner. However this causes to abort the session. All of the services that are executed up to the abort decision have world altering effects in this scenario. Therefore their calls are conducted through the business activity coordinator. After the abort decision is made, the software agent sends “compensateAll” signal to the coordinator which then transmits it to the

participants. The participant services fire their compensation mechanisms. As a result, undesired side effects of the previously executed actions are prevented.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

Web becomes more important for human beings day by day. Humans perform important amount of B2B and B2C operations using functionalities provided on web. Those functionalities are generally provided by means of web services. Web services provide operating system and programming language neutral environment which enables interoperability between distinct systems so they are widely used in real world. Although many ready to use web services exist, it is very difficult for human beings to analyze their functionalities and possible collaborations among them. As a result, this process should be automatically handled. Efforts on semantic web intend to make web machine interpretable and enables the automation of composing web services. This thesis proposed a novel solution for performing automated web service composition and invocation. Users present their request to the proposed software agent and the agent handles the required analysis. It finds required web services, the execution order of found web services and possible data transfers among them. In the literature, a considerable amount of work on automated WSC problem has been conducted, but still there exist many open issues. This thesis proposed solutions for some of the open issues by using the features of an AI planner namely "SIMPLANNER" and it also proposed integrating the web service transaction frameworks (i.e. WS-Coordination and WS-Business Activity) to its automatic web service invocation mechanism. The solution that is proposed by this thesis is highly adaptable which makes it appropriate for real word applications.

The important features of the proposed "automated web service composition and invocation framework" can be summarized as follows:

- It is highly fault tolerant which is very important for real world applications.

The proposed system interleaves planning and execution. If some problems occur during service execution such as service unavailability because of network problems, wrongly provided inputs, etc. or information unavailability that is the user or another web service cannot provide the required inputs for services, the unexpected event handler is fired. The unexpected event handler initially tries to resolve the problem in a high level by using the dynamic replanning features of Simplanner. If it cannot solve the problem in high level some physical operations are conducted. Since world altering service calls are done by conforming the WS-Business Activity specification, in failure situations compensation mechanisms are fired which prevent undesired side effects.

- It is responsive; users do not to wait for long periods of time for getting the results of their requests.

The component that determines the time for responding to the user is the AI planner, since computationally complex operations are handled by it. One of the design principles of Simplanner is its timely response. It is designed for real time operations. Simplanner is any time planner. It concentrates not on the whole plan but on the first action. It finds the first best action in polynomial time and during real execution it finds the next action. Deliberation time required by the planner is too short, and sometimes this short deliberation time can be eliminated. The system has an action caching mechanism. If a similar problem is previously solved by the software agent, precompiled solutions are directly used. Shortly, the proposed system is highly responsive.

- Users do not need to provide excessive amount of information initially; the software agent asks only the required ones during the composition process.

It is impossible for users to know which services are going to be used for handling their requests, so they cannot provide all required inputs initially. In the proposed system, users initially give a very high level description of their

requests. The software agent discovers the required services for the high level definitions and asks the user the required input. If users cannot provide the input, the software agent discovers new paths that do not require that particular information or if possible, discovers some other web services that can provide that particular information.

- Dynamic object generation of web services is modeled by using some in-memory data structures and with continuous user interactions.

Web services sometimes produce information which is not available initially. This information is represented as objects in the planning domain and it is very difficult to represent dynamic objects in AI planning. In this system, the software agent constructs an object for each type definition of the worked domain. Their availability and unavailability is represented by using some logical statements and determined according to user interactions.

This thesis provided these important contributions to the automated web service composition and invocation problem. However, some important future work still exists. The most important future work is providing scalability. Almost all domain independent AI planners fail to work with more than thousands of actions which is a very small number for real world cases. One of the possible solutions is to do some filtering before stating the problem to the planner. Filtering operation will eliminate irrelevant web services according to the user's goal and give the planner a problem with a reasonable search space. Filtering may eliminate some relevant services as well but it is acceptable, otherwise the proposed system cannot be used in real world where there exist millions of web services.

Another future work is to include some more syntactic analysis to the system. In the current system, sometimes inputs of the service cannot be provided by the users but by some other services. In that case, semantic types are compared. However semantic type similarity does not mean that syntactic types are equal all the time. For instance, the output of service A provide the input of service B, that is; the output of A and the input of service B has same semantic type, but their WSDL

counterparts may be different. Such cases are rare but they are problematic situations which should be solved by some syntactic analysis.

Another future work is to involve the user to the WSC procedure more. The user can see the results of the executed services immediately and according to those results they might be able to direct the software agent. If a service returns an undesired result, they will be able to invalidate that service for that session through some interfaces. In the current system, service invalidations are conducted automatically for some unexpected situations. This mechanism can easily be used by the users themselves manually when desired.

REFERENCES

[1] Rao J., Su X. A Survey of Automated Web Service Composition Methods. Proceedings of 1sth International Workshop on Semantic Web Services and Web Process Composition, pages 43-54, 2004.

[2] Christensen E., Curbera F., Meredith G., Weerawarana S. Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/wsdl, last visited on 21.07.2009.

[3] Smith M.K., Welty C., McGuinness D. L. OWL Web Ontology Language Guide, http://www.w3.org/TR/owl-guide/, last visited on 21.07.2009.

[4] Martin D., Burstein M., Hobbs J., Lassila O., McDermott D., McIlraith D., Narayanan S., Paolucci M., Parsia B., Payne T., Sirin E., Srinivasan N., Sycara K. OWL-S: Semantic Markup for Web Services, http://www.w3.org/Submission/OWL-S/, last visited on 21.07.2009.

[5] Ghallab M., Howe A., Knoblock C., McDermott D., Ram A., Veloso M., Weld D., Wilkins D. PDDL: The Planning Domain Definition Language, AIPS-98 Planning Committee, 1998.

[6] Sapena O., Onaindia E. Planning in Highy Dynamic Environments: An Anytime Approach for Planning Under Time Constraints. Journal of Applied Intelligence, Volume 29, Number 1, pages 90-109, August 2007.

[7] OASIS Web Services Business Activity Specification, http://docs.oasis-open.org/ws-tx/wsba/2006/06, last visited on 21.07.2009.

[8] Milanovic N., Malek, M. Current Solutions for Web Service Composition. IEEE Transactions on Internet Computing, Volume:8, Issue:6, pages 51-59, 2004.

[9] Srivastava B., Koehler J. Web Service Composition – Current Solutions and Open Problems. ICAPS 2003 Workshop on Planning for Web Services, 2003.

[10] Polleres A. AI Planning For Web Service Composition. Presentation, Ilog, Paris, France, 2004. http://axel.deri.ie/~axepol/presentations/20040907-paris-ilog-AIplanning4WSC.ppt, last visited on 21.07.2009.

[11] Agarwal V., Chafle G., Mittal S., Srivastava B. Understanding Approaches for Web Service Composition and Execution, IBM Research Report, August 2007.

[12] OASIS Web Services Coordination Specification, http://docs.oasis-open.org/ws-tx/wscoor/2006/06, last visited on 21.07.2009.

[13] OASIS Web Services Atomic Transaction Specification, http://docs.oasis-open.org/ws-tx/wsat/2006/06, last visited on 21.07.2009.

[14] Haas H., Brown A. Web Services Glossary, http://www.w3.org/TR/ws-gloss/, last visited on 21.07.2009.

[15] Box D., Ehnebuske D., Kakivaya G., Layman A., Mendelsohn N., Nielsen H. F., Thatte S., Winer D. Simple Object Access Protocol (SOAP) 1.1, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/, last visited on 21.07.2009.

[16] Bellwood T., UDDI Version 2.04 API Specification, http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm#_Toc25137692, last visited on 21.07.2009.

[17] Web Service Modeling Ontology (WSMO), http://www.wsmo.org/, last visited on 21.07.2009.

[18] Peer J. Semantic Service Markup with SESMA. Proceedings of International World Wide Web Conference 2005, 2005.

[19] Russel S., Norvig P. Artificial Intelligence: A Modern Approach, 3rd edition, 2003.

[20] Blum A., Furst M., Fast Planning Through Planning Graph Analysis. Proceedings of 14th International Joint Conference on Artificial Intelligence, pages 1636-1642, 1995.

[21] STRIPS language, http://en.wikipedia.org/wiki/STRIPS, last visited on 21.07.2009.

[22] ADL language, http://en.wikipedia.org/wiki/Action_description_language, last visited on 21.07.2009.

[23] Helmert M., An Introduction To PDDL, http://www.cs.toronto.edu/~sheila/2542/w09/A1/introtopddl2.pdf, last visited on 21.07.2009.

[24] Peer J. Web Service Composition as AI Planning – a Survey. Technical report, Univ. of St. Gallen, 2005.

[25] Casati F., Ilnicki S., Jin L. Adaptive and Dynamic Service Composition in EFlow. Proceedings of 12th International Conference on Advanced Information Systems Engineering, 2000.

[26] Schuster H., Georgakopoulos D., Cichocki A., Baker D. Modeling and Composing Service-Based and Reference Process-Based Multi-Enterprise Processes. Proceedings of 12th International Conference on Advanced Information Systems Engineering, 2000.

[27] Sirin E., Parsia B., Wu D., Hendler J., Nau D., HTN Planning for Web Service Composition Using SHOP2, Journal of Web Semantics, pages 377-396, 2004.

[28] Nau D., Au T.C., Ilghami O., Kuter U., Murdock W., Wu D., Yaman F. SHOP2: An HTN Planning System, JAIR Volume 20, pages 379-404, 2003.

[29] M. Klusch, A. Gerber, M. Schmidt. Semantic Web Service Composition Planning with OWLS-XPlan. Proceedings of the AAAI Fall Symposium on Semantic Web and Agents, Arlington VA, USA, AAAI Press, 2005.

[30] Hoffmann, J. The Metric-FF planning system: Translating Ignoring Delete Lists to Numeric State Variables. Journal of Artificial Intelligence Research (JAIR) vol 20, 2003.

[31] Peer J. A PDDL Based Tool for Automatic Web Service Compostion. Proceedings of 2nd International Workshop on Principles and Practice of Semantic Web Reasoning, pages 149-163, 2004.

[32] WSPlan, http://sourceforge.net/projects/wsplan/, last visted on 21.07.2009.

[33] Younes, H.L.S., Simmons, R.G.: VHPOP: Versatile Heuristic Partial Order Planner. Journal of Artificial Intelligence Research (JAIR), 2003.

[34] Gerevini, A., Saetti, A., Serina, I: Planning through Stochastic Local Search and Temporal Action Graphs. Journal of Artificial Intelligence Research, 2003.

[35] Agarwal V., Dasgupta K., Karnik N., Kumar A., Kundu A., Mittal S., Srivastava B. A Service Creation Environment Based on End to End Composition of Web Services, Proceedings of the 14th international conference on World Wide Web, 2005.

[36] WS-BPEL, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel, last visited on 21.07.2009.

[37] Srivastava B. A Software Framework for Building Planners. Proceedings of Knowledge Based Computer Systems, 2004.

[38] Java Reflection, http://java.sun.com/docs/books/tutorial/reflect/index.html, last visited on 21.07.2009.

[39] Kim H., Kim I., Mapping Semantic Web Service Descriptions to Planning Domain Knowledge. Proceedings of IFMBE, Volume 14, pages 388-391, Springer Berlin Heidelberg, July 2007.

[40] OWLS2PDDL tool, http://projects.semwebcentral.org/projects/owls2pddl/, last visited on 21.07.2009.

[41] Bryce D., Kambhampati S. A Tutorial on Planning Graph-Based Reachability Heuristics, AI Magazine, Vol 28, No 1, 2007.

[42] WSIF, Web Services Invacation Framework, http://ws.apache.org/wsif/, last visted on 21.07.2009.

[43] Axis, Apache Web Services Project, http://ws.apache.org/axis/, last visited on 21.07.2009.

[44] Kandula, Apache WS-Transaction Project, http://ws.apache.org/kandula/, http://ws.apache.org/axis/, last visited on 21.07.2009.

[45] Freund T., Storey T. Transactions in the world of Web services, Part 1. An overview of WS-Transaction WS-Coordination, http://www.ibm.com/developerworks/webservices/library/ws-wstx1/, last visited on 21.07.2009.

[46] Freund T., Storey T. Transactions in the world of Web services, Part 2. An overview of WS-Transaction WS-Coordination, http://www.ibm.com/developerworks/webservices/library/ws-wstx2/, last visited on 21.07.2009.

[47] Erven H., Hicker G., Huemer C., Zaptletal M. The Web Services-BusinessActivity-Initiator (WS-BA-I) Protocol: an Extension to the Web Services-BusinessActivity Specification. IEEE International Conference on Web Services 2007, 2007.

[48] OWLS-XPLAN, http://projects.semwebcentral.org/projects/owls-xplan/, last visted on 21.07.2009.

[49] Berners-Lee T., Hendler J., Lassila O., The Semantic Web, Scientific American Magazine, 2001.

[50] Kuster U., Stern M., Konig-Ries B., A Classification of Issues and Approaches in Automativ Service Composition. 1sth Intermational Workshop on Engineering Service Compositions, 2005.

[51] Kuter U, Sirin E., Parsia B., Nau D., Hendler J. Information Gathering During Planning for Web Service Composition. Proceedings of ICAPS-P4WGS 2004, 2004.

[52] Oh S.C., Lee, D., Kumara S. A Comparative Illustration of AI Planning based Web Service Composition, ACM Sigecom Exchanges, Volume 5, pages 1-10, 2006.

[53] Pistore M., Bertoli P., Barbon F., Shaparau D., Traverso P. Planning and Monitoring Web Service Composition. Proceedings of 14th International Conference on Automated Planning and Scheduling, 2004.

[54] Sheshagiri M. Automatic Composition and Invocation of Semantic Web Services, MS Thesis, Faculty of the Graduate School of the University of Maryland, 2004.

[55] Zhang R. Ontology Driven Web Services Composition Techniques. MS. Thesis, Faculty of the Graduate School of The University of Georgia, 2004.

[56] Sirin E., Hendler J., Parsia B. Semi-automatic Composition of Web Services Using Semantic Descriptions. Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2003, 2003.

[57] Rao D., Jiang Z., Jinag Y. Fault Tolerant Web Service Composition as Planning. Proceedings of International Conference on Intelligent Systems and Knowledge Engineering 2007, 2007.

[58] Sheshagiri M., Desjardins M., Finin T. A Planner for Composing Services Described in DAML-S. Proceedings of the AAMAS Workshop on Web Services and Agent-based Engineering, 2003.

[59] Onaindia E., Sapena O., Sebastia L., Marzal E. SimPlanner: An Execution-Monitoring System for Replanning in Dynamic Worlds. Lecture Notes in Computer Science, Progress in Artificial Intelligence, Volume 2258, pages 393-400, Springer Berlin Heidelberg, 2001.

[60] JUNG, Java Universal Network/Graph Framework, http://jung.sourceforge.net/, last visted on 21.07.2009.

APPENDIX A

TRAVEL ONTOLOGY

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns="http://127.0.0.1/health-scallops/ontology/Health-
Scallops_Ontology.owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
  xml:base="http://127.0.0.1/health-scallops/ontology/Health-
Scallops_Ontology.owl">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="Transport"/>
  <owl:Class rdf:ID="Airport"/>
  <owl:Class rdf:ID="ArrivalAirport">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Airport"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="DepartureAirport">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Airport"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Flight">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:allValuesFrom>
          <owl:Class rdf:about="#Airport"/>
        </owl:allValuesFrom>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#hasDepartureLocation"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
```

```
<rdfs:subClassOf>
      <owl:Restriction>
        <owl:allValuesFrom>
          <owl:Class rdf:ID="FlightParameters"/>
        </owl:allValuesFrom>
        <owl:onProperty>
          <owl:ObjectProperty rdf:resource="#hasParameters"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty
rdf:resource="#hasDestinationLocation"/>
        </owl:onProperty>
        <owl:allValuesFrom>
          <owl:Class rdf:about="#Airport"/>
        </owl:allValuesFrom>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="MedicalTreatment"/>
  <owl:Class rdf:about="#Airport">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Location"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Creditcard"/>
  <owl:Class rdf:ID="MedicalFlightParameters">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#FlightParameters"/>
    </rdfs:subClassOf>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty
rdf:resource="#assuresMedicalTreatment"/>
        </owl:onProperty>
        <owl:someValuesFrom>
          <owl:Class rdf:resource="#MedicalTreatment"/>
        </owl:someValuesFrom>
      </owl:Restriction>
    </owl:equivalentClass>
    <owl:disjointWith>
      <owl:Class rdf:about="#FlightParameters"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:about="#FlightParameters">
    <owl:disjointWith rdf:resource="#MedicalFlightParameters"/>
    <rdfs:subClassOf>
      <owl:Class rdf:ID="TransportParameters"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Category"/>
  <owl:Class rdf:ID="Account"/>
  <owl:Class rdf:ID="Time"/>
  <owl:Class rdf:ID="FlightCategory">
```

```
  <rdfs:subClassOf rdf:resource="#Category"/>
</owl:Class>
<owl:Class rdf:ID="Address"/>
<owl:Class rdf:ID="VehicleTransport">
  <rdfs:subClassOf rdf:resource="#Transport"/>
</owl:Class>
<owl:Class rdf:ID="Patient">
  <rdfs:subClassOf>
    <owl:Class rdf:ID="Person"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="VehicleTransportParameters">
  <rdfs:subClassOf rdf:resource="#TransportParameters"/>
  <owl:disjointWith>
    <owl:Class rdf:ID="MedicalVehicleTransportParameters"/>
  </owl:disjointWith>
</owl:Class>
<owl:Class rdf:ID="Hospital">
  <rdfs:subClassOf rdf:resource="#Location"/>
</owl:Class>
<owl:Class rdf:about="#MedicalVehicleTransportParameters">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#assuresMedicalTreatment"/>
      </owl:onProperty>
      <owl:someValuesFrom rdf:resource="#MedicalTreatment"/>
    </owl:Restriction>
  </owl:equivalentClass>
  <rdfs:subClassOf rdf:resource="#VehicleTransportParameters"/>
</owl:Class>
<owl:Class rdf:ID="Company"/>
<owl:Class rdf:ID="Vehicle"/>
<owl:Class rdf:ID="ProvidedFlight">
  <rdfs:subClassOf rdf:resource="#Flight"/>
</owl:Class>
<owl:Class rdf:ID="ProvidedTransport">
  <rdfs:subClassOf rdf:resource="#VehicleTransport"/>
</owl:Class>
<owl:Class rdf:ID="ValidAccount">
  <rdfs:subClassOf rdf:resource="#Account"/>
</owl:Class>
<owl:ObjectProperty rdf:ID="isBookedFor">
  <rdfs:domain rdf:resource="#Transport"/>
  <rdfs:range rdf:resource="#Person"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="assuresMedicalTreatment">
  <rdfs:domain rdf:resource="#TransportParameters"/>
  <rdfs:range rdf:resource="#MedicalTreatment"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasCategory">
  <rdfs:domain rdf:resource="#FlightParameters"/>
  <rdfs:range rdf:resource="#FlightCategory"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasParameters">
  <rdfs:domain rdf:resource="#Transport"/>
  <rdfs:range rdf:resource="#TransportParameters"/>
</owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:ID="hasDepartureLocation">
  <rdfs:domain rdf:resource="#Transport"/>
  <rdfs:range rdf:resource="#Location"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasArrivalTime">
  <rdfs:domain rdf:resource="#TransportParameters"/>
  <rdfs:range rdf:resource="#Time"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasDestinationLocation">
  <rdfs:domain rdf:resource="#Transport"/>
  <rdfs:range rdf:resource="#Location"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="isOwnedByPerson">
  <rdfs:domain rdf:resource="#Creditcard"/>
  <rdfs:range rdf:resource="#Person"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="isOwnedByCompany">
  <rdfs:domain rdf:resource="#Creditcard"/>
  <rdfs:range rdf:resource="#Company"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasDepartureTime">
  <rdfs:domain rdf:resource="#TransportParameters"/>
  <rdfs:range rdf:resource="#Time"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="validPersonalFlightAccount">
  <rdfs:range rdf:resource="#Account"/
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="validMedicalFlightAccount">
  <rdfs:range rdf:resource="#Account"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="medicalFlightParameters">
  <rdfs:range rdf:resource="#FlightParameters"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="personalProvidedFlight">
  <rdfs:range rdf:resource="#Flight"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="medicalProvidedFlight">
  <rdfs:range rdf:resource="#Flight"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="medicalProvidedTransport">
  <rdfs:range rdf:resource="#VehicleTransport"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="medicalVehicleParameters">
  <rdfs:range rdf:resource="#VehicleTransportParameters"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="personalProvidedTransport">
  <rdfs:range rdf:resource="#VehicleTransport"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="validPersonalTransportAccount">
  <rdfs:range rdf:resource="#Account"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="validMedicalTransportAccount">
  <rdfs:range rdf:resource="#Account"/>
</owl:ObjectProperty>
<owl:FunctionalProperty rdf:ID="hasNearestAirport">
  <rdfs:domain rdf:resource="#Address"/>
  <rdfs:range rdf:resource="#Airport"/>
</owl:FunctionalProperty>
```

```
  <owl:FunctionalProperty rdf:ID="usesVehicle">
    <rdfs:domain rdf:resource="#VehicleTransportParameters"/>
    <rdfs:range rdf:resource="#Vehicle"/>
  </owl:FunctionalProperty>
  <owl:FunctionalProperty rdf:ID="hasAddressLocation">
    <rdfs:domain rdf:resource="#Location"/>
    <rdfs:range rdf:resource="#Address"/>
  </owl:FunctionalProperty>
  <owl:FunctionalProperty rdf:ID="hasAddressPerson">
      <rdfs:domain rdf:resource="#Person"/>
     <rdfs:range rdf:resource="#Address"/>
  </owl:FunctionalProperty>
  <owl:FunctionalProperty rdf:ID="hasAddressCompany">
      <rdfs:domain rdf:resource="#Company"/>
     <rdfs:range rdf:resource="#Address"/>
  </owl:FunctionalProperty>
</rdf:RDF>
```