A MONOLITHIC APPROACH TO AUTOMATED COMPOSITION OF SEMANTIC WEB SERVICES WITH THE EVENT CALCULUS

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

BY

ÇAĞLA OKUTAN

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

SEPTEMBER 2009

Approval of the thesis:

A MONOLITHIC APPROACH TO AUTOMATED COMPOSITION OF SEMANTIC WEB SERVICES WITH THE EVENT CALCULUS

submitted by **ÇAĞLA OKUTAN** 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. Ferda Nur Alpaslan
Computer Engineering Dept., METU _____

Dr. Ayşenur Birtürk
Computer Engineering Dept., METU _____

Dr. Orkunt Sabuncu
ORBIM, METU _____

Date: _____

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: ÇAĞLA OKUTAN

Signature :

ABSTRACT

A MONOLITHIC APPROACH TO AUTOMATED COMPOSITION OF SEMANTIC WEB SERVICES WITH THE EVENT CALCULUS

Okutan, Çağla
M.S., Department of Computer Engineering
Supervisor : Assoc. Prof. Dr. Nihan Kesim Çiçekli

September 2009, 70 pages

In this thesis, a web service composition and execution framework is presented for semantically annotated web services. A monolithic approach to automated web service composition and execution problem is chosen, which provides some benefits by separating the composition and execution phases. An AI planning method using a logical formalism called Event Calculus is chosen for the composition phase. This formalism allows one to generate a narrative of actions and temporal orderings using abductive planning techniques given a goal. Functional properties of services, namely input/output/precondition/effects(IOPE) are taken into consideration in the composition phase and non-functional properties, namely quality of service (QoS) parameters are used in selecting the most appropriate solution to be executed. The repository of OWL-S semantic Web services are translated to Event Calculus axioms and the resulting plans found by the Abductive Event Calculus Planner are converted to graphs. These graphs can be sorted according to a score calculated using the defined quality of service parameters of the atomic services in the composition to determine the optimal solution. The selected graph is converted to an OWL-S file which is executed consequently.

Keywords: Automatic Web Service Composition, Semantic Web Services, Event Calculus, Execution of Composite Web Services, OWL-S

ÖZ

ANLAMSAL WEB SERVİS BİLEŞİMİ BELİRLEME İŞLEMİNİN EVENT CALCULUS KULLANILARAK TEKİL BİR YÖNTEMLE OTOMATİKLEŞTİRİLMESİ

Okutan, Çağla
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Assoc. Prof. Dr. Nihan Kesim Çiçekli

Eylül 2009, 70 sayfa

Bu tezde, anlamsal örün servisleri için bir birleşim ve yürütme sistemi sunulmuştur. Birleşim ve yürütme işlevlerini ayırmasıyla bir takım yararlar sağlayan tekil bir yöntem izlenmiştir. Mantıksal bir sistem sunan Event Calculus kullanılarak birleşim problemi yapay zeka planlama problemine çevrilmiştir. Bu mantıksal sistem, bir hedef için, açıklama bulmaya yönelik teknikler kullanılarak, olay ve zaman sıralamalarından oluşan planlar üretilmesini mümkün kılmaktadır. Fonksiyonel servis özellikleri yani girdi/çıktı/önkoşul/sonuç'lar birleşim aşamasında, fonksiyonel olmayan servis özellikleri yani servis kalite parametreleri ise yürütülecek en uygun birleşimi seçme aşamasında kullanılır. OWL-S havuzunda bulunan örün servisleri Event Calculus aksiyomlarına çevrilir ve Abductive Event Calculus Planner'ın bulduğu sonuçlar grafiklere dönüştürülür. Bu grafikler, birleşimlerdeki servislere ait tanımlanmış servis kalite parametreleri kullanılarak hesaplanan bir puana göre sıralanabilir. Seçilen grafik daha sonra yürütülecek olan OWL-S dosyasına çevrilir.

Anahtar Kelimeler: Otomatik Örün Servis Birleşimi, Anlamsal Örün Servisleri, Çıkarımsal Olay Cebiri, Örün Servis Birleşimi İşletilmesi, OWL-S

To my family

ACKNOWLEDGMENTS

I would like to thank in the first place to Assoc. Prof. Dr. Nihan Kesim Çiçekli who has given any kind of support one may want from her supervisor during this thesis study. Her understanding, caring, knowledge and well-organization always brightened my way.

I would like to thank Mehmet Kuzu who has been working in a similar subject, has helped me gain different viewpoints and has eased the job of understanding new concepts by his valuable discussions.

I would like to thank my family for their never ending support and concern.

I would like to thank Hande Çelikkanat and Bahar Pamuk, who have never withhold their words to raise me up when I was in my down mood.

I would like to thank the Scientific and Technological Research Council of Turkey (TÜBİTAK) for supporting me financially during the first terms of this study.

As a final acknowledgment, I would like to thank my colleagues nicknamed 'ekip' in TÜBİTAK, whose endless help in everything has been very valuable for me.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

Web services are pivotal point in developing applications on today's web since they allow fast delivering of new software. The traditional programming is often done in an ad hoc manner and quite often the reusability is restricted. Web services on the other hand give an extensive opportunity for software integration and reusability. Current standards for Web services are WSDL, SOAP, BPEL and UDDI. WSDL is an XML-based document that describes a Web service, how to access the service by specifying the location of the service and the operations the service exposes. SOAP is an XML-based protocol for service messaging. BPEL is a standard for defining Web service interactions and composition flows. UDDI is an XML-based registry which makes services discoverable over Internet. Apart from presenting low-level agreements between Web services and providing a distributed environment, these standards require a lot of human effort to create and manage end-to-end service interactions. In such a dynamic world, where services are created, updated, deleted on the fly, these industry standards are far from being sufficient.

Academia solution for giving Web services a *meaning*, so that besides humans, software agents can also benefit from, is by means of semantic web issues, that is annotating Web services with OWL-S which is an ontology built on top of Web Ontology Language(OWL). This method presents a way for giving an abstract representation of Web services which can be characterized with functional properties like input/output/precondition/effect(IOPE) and also some non-functional properties such as availability, execution duration, reliability which are called Quality of Service(QoS) parameters also.

Using AI planning techniques for Web service composition problem is a long study in academia. The composition problem is transferred into a planning problem by translating Web service

capabilities defined in ontologies and searched for solutions for the problem by using goal-oriented inferencing techniques from planning. AI planning methods are mainly concerned with the functional properties, IOPEs of services to express in planning domain. Operations of the planning domain (actions or events) become operations of the composition domain (Web services). Inputs and outputs of a Web service become knowledge precondition and knowledge effects respectively. Also preconditions and effects of Web services are transferred to preconditions and effects of actions in planning domain. The composition problem is mainly to reach the goal state from the initial state. The literature includes many research on applying AI planning techniques to Web service composition domain [1, 2, 3, 4, 5, 6, 7, 8, 9]. These proposals mostly come up with an abstract composition that needs further manual intervention for actually creating the deployable and executable service.

The need for a tool for Web service composition seems to increase in great amounts. Today there are many Web services that are ready to be discovered and used on their own. For illustrating a simple scenario, a person needs the information about the price of this book by starting from a book name. There are services where one can find the actual book information with the ISBN number, author name and publisher information where book name is known and one can find the price of the book where ISBN number is known. Given a fixed set of Web services, using only WSDL descriptions to compose these two services would comprise of manually analyzing the services and seeing if they suit the purpose. With ontologies defined for Web services and a composition tool in hand, the composition problem reduces to determining the known inputs and wanted outputs. The automation of creating a business process functionality based on requirements from user, developing executable workflows and deploying them on an execution environment become possible.

Planning with AI techniques focuses on capability matching: the matching between IOPE of advertised services and IOPE of requester services. In complex problems, or in cases where some properties of Web services become important to the user, a selection mechanism is needed to find the optimal plan. For this purpose QoS parameters are used. There are many algorithms in the literature that try to choose the best composition plan by optimizing the calculated score for each service based on QoS parameters[10, 11, 12, 13, 14, 15, 16, 4]. This approach gives the user more control on selecting the most appropriate plan especially if the plan is composed of many services, and there are alternative services that have the same capability matching but different quality values.

This thesis aims to propose a framework which binds semantically-annotated Web service components together and delivers the required function given user's specifications for a new service. The work of [1] is extended to handle execution of the generated composed plan and to sort plans for best plan selection. OWL-S is used for semantically describing Web services. The AI planning technique involves the use of Event Calculus (EC), a logical framework for expressing actions and their effects. What Event Calculus does in Shanahan's[17] words is: "The event calculus is a logical mechanism that infers *what's true when* given *what happens when* and *what actions do*. The *what happens when* part is a narrative of events, and the *what actions do* part describes the effects of actions.". Using the axioms of the defined formalism, *what happens when* part is tried to be inferred by using Abductive Event Calculus Planner[18].

Event Calculus provides the necessary platform for encoding the Web service composition problem as an AI planning problem. It has representational efficiency, with capability of representing concurrent actions, actions with indirect and non-deterministic effects, compound actions and continuous change. It provides an abstractness and does not require domain knowledge specific to any kind of problem. Also with Event Calculus, it is possible to express the goals as complex expressions.

IOPEs extracted from OWL-S descriptions of Web services are transformed to the planner's domain and using the axioms of event calculus, the planner returns a narrative of actions as plans which are possibly total or partial order. The output of the generated solution is transformed into a graph object, which is in the first place visually presented to the user. The framework allows the user to sort the generated graphs according to some aggregation value calculated based on QoS parameters. The selected graph object is parsed and transformed into a new OWL-S service which has the functionality determined by the user at initial step. The file is executed by the user specified input values using the execution engine, and results are displayed to the user. The OWL-S services used by the planner is a fixed set contained in a repository. That is, Web service discovery problem is out of the scope of this thesis.

The organization of the work is as follows: In the subsequent chapter, background information and a survey of existing literature on AI planning techniques on Web service composition problem are presented. In Chapter 3, the general architecture of the proposed framework is presented. In Chapter 4, the details of using QoS parameters in plan selection phase and how preconditions and effects are handled in both planing and execution phases are given.

Chapter 5 illustrates the use of the system with a case scenario by giving details about the implementation. Chapter 6 concludes with some evaluation together with pointing future extensions to the proposed system.

CHAPTER 2

BACKGROUND INFORMATION AND RELATED WORK

This chapter aims for representing background information on Web services and related technologies, Event Calculus formalism, how abductive planning is done with Event Calculus, and current studies related to Web service composition problem. The basic concepts, thoroughly used terminologies and definitions are given.

2.1 Background Information on Web Services

2.1.1 Web Services

Components are the building blocks of an application environment in traditional programming. Much like Web services are components of a service-oriented architecture which have numerous of unique characteristics. The complete autonomic behavior of a Web service from another is one such quality and provides loosely coupling in-between. Each service is responsible for its own domain and implementing a set of business functions. The isolated environments are bound by a common agreement to a standard communications framework. Each service independently has the right to decide its own programming logic, platform and technology. By encapsulating the function definition, by loosely coupling and having a contract, web services provide the general structure for interoperability. The communication protocols provide the chance of requesters and providers to be in distributed environments. Web services must conform to some standards. Fig 2.1 shows the relationship between these standards.

WSDL In order to be discovered and interfaced with other services, a web service has to be

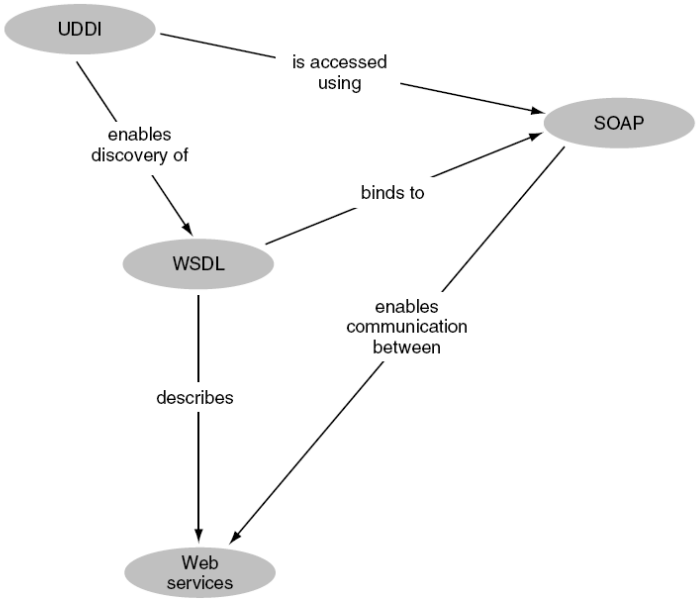


Figure 2.1: Relationships between standards

defined in a consistent manner. The Web Services Definition Language[19] is a World Wide Web Consortium (W3C) specification providing the language for description of Web service definitions. WSDL enables communication between the integration layers by defining endpoint descriptions.

SOAP The messaging between Web service peers in a distributed environment is standardized by Simple Object Access Protocol [20]. The exchanged XML-based information structure is defined with this standard.

UDDI A requester web service requires discovery of the needed provider service. A central directory that hosts service descriptions is needed. The Universal Description, Discovery, and Integration[21] specification is the global registry which provides a standardized way of publishing and discovering of Web services along the Web.

These standards are basic building blocks for Web service framework but richer semantic specifications of Web services are needed to enhance the power of the framework as discussed in the following section.

2.1.1.1 Web Services Composition Problem

Web service composition problem may be expressed as selecting and combining Web services to achieve a user's goal. It is far beyond the human capability to make the composition manually in the current evolving Web where any time a new service is created and disappears. For automating Web service composition problem, it is required to express all the information in an unambiguous computer interpretable form. The current industrial standards which are mentioned before do not handle Web service preconditions and (conditional) effects nor provides an unambiguous way of mapping inputs and outputs. In addition to mapping input/output/precondition/effect (IOPEs), composition problem involves selecting the best Web service among alternatives by the use of properties, capabilities and functioning of a Web service. The use of semantic web features becomes important in order to satisfy these requirements. Web service composition problem can be both seen as a synthesis and as an AI planning problem. In the former approach, an abstract workflow is executed based on user constraints. In the latter approach, the domain descriptions are transformed to planning domain and a planner is used for generating the plan using the goal description. Both approaches are further investigated in Section 2.4.

Web service composition is a research topic which contains many challenges and discussions. These can be listed as nondeterministic behavior of web services that causes failures during execution like unavailability of a registered service and network failures, partial observability, scalability and handling of services which have effects changing the world state. Nevertheless, a semantic approach combined with AI planning techniques seems to have a solution to the problems that current standards are not capable of solving.

2.1.2 OWL-S

Manually programming and designing Web services is a tedious and costly job in the current evolving Web. A huge time is elapsed while searching the right service, adapting components between incompatible ones. Web Services Choreography Description Language (WS-CDL) proposed by W3C is aimed at solving the problem of collaboration regardless of the supporting platforms on each side. Business Process Execution Language for Web Services(BPEL4WS) [22] is another language specification for formal specification of business

processes and business interaction protocols. But these specifications also are harder to deal with and do not provide enough abstraction and comfort. Semantic Web technologies will contribute to Web service frameworks as well. With richer semantics it is possible to [23, 24]:

- automate service selection, composition, verification and invocation
- automate translation of message content between services
- handle service monitoring and recovery from failure in a more comprehensive manner
- reduce manual configuration and programming efforts.

Semantic Web services, describe the concepts in the services' domains by the help of ontologies, some characteristics of services like control and data flow and service's relationships to the domain ontologies by the help of IOPEs. Currently popular semantic Web service specifications are Web Service Modeling Ontology (WSMO)[25] and OWL-S[26]. Literature survey shows that OWL-S is used more widely in web service composition with AI planning. So in this thesis, OWL-S definitions of services are used.

Formal definition of OWL-S is given in [26] as "OWL-S is an ontology, within the OWL-based framework of the Semantic Web, for describing Semantic Web Services. It will enable users and software agents to automatically discover, invoke, compose, and monitor Web resources offering services, under specified constraints." Description of a service is mainly defining the *profile*, *process model* and *grounding* parts of the *service* each of which is explained below:

profile This part is used to describe what the service does. A profile provides a general description of a Web service, and includes functional properties (IOPEs) and nonfunctional properties like category, quality rating and additional service parameters. This is the part mainly used for human reading and web service discovery. The functional properties may be a subset of the ones explained in process model. Figure 2.2 shows a *profile* for a book finder service describing that the the service has an input parameter named *BookName*, an output of parameter named *Book* and a precondition named *AlphanumericCheck*:

process model This part is responsible for describing how a service performs its tasks. It includes information about inputs, outputs and under which conditions the outputs will

```
<profile:Profile rdf:ID="BookFinderProfile">
    <service:presentedBy rdf:resource="#BookFinderService"/>

    <profile:serviceName xml:lang="en">Book Finder</profile:serviceName>
    <profile:textDescription xml:lang="en">This service returns the
information of a book whose title best matches the given string.
    </profile:textDescription>

    <profile:hasInput rdf:resource="#BookName"/>

    <profile:hasOutput rdf:resource="#Book"/>

    <profile:hasPrecondition rdf:resource="#AlphanumericCheck"/>

</profile:Profile>
```

Figure 2.2: Book Finder Service Profile

occur, preconditions which are circumstances that must hold before a service can be used, and results which are changes the service has caused. There are three types of processes: composite, atomic, and simple processes. Composite processes are formed by simpler component processes that are bound to each other by data bindings which provide data flow among each other and control construct that provides the control flow. Atomic processes present directly invokable functional property. Simple processes are abstract process descriptions which are not invokable but may relate to other composite or atomic processes. The control constructs of OWL-S are Sequence, Split, Split + Join, Choice, Unordered, Condition, If-Then-Else, Iterate, Repeat-While, and Repeat-Until. In Figure 2.3, the mentioned properties and their relationships within each other can be seen.

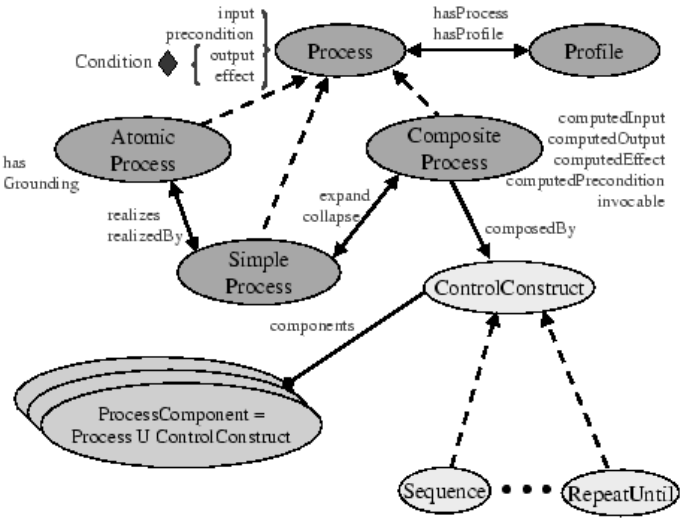


Figure 2.3: OWL-S Process Model

Preconditions and effects are specified as logical formulas and OWL-S does not dictate any language for expressing them. Semantic Web Rule Language (SWRL)[27], Resource Description Framework (RDF)[28], Knowledge Interchange Format (KIF)[29] and Planning Domain Definition Language (PDDL)[30] are alternative languages. Among them SWRL is the mostly used one. SWRL is "based on a combination of the OWL DL and OWL Lite sublanguages of the OWL Web Ontology Language with the Unary/Binary Datalog RuleML sublanguages of the Rule Markup Language." as stated in [27]. Preconditions and effects are expressed by the *Expression* property of process. An example process precondition can be seen in Figure 2.4:

```
<process:AtomicProcess rdf:ID="BabelFishTranslatorProcess">
  ...
  <process:hasPrecondition rdf:resource="#SupportedLanguagePair"/>
</process:AtomicProcess>

<expr:SWRL-Condition rdf:ID="SupportedLanguagePair">
  <rdfs:label>canBeTranslatedTo(InputLanguage, OutputLanguage)</rdfs:label>
  <expr:expressionLanguage rdf:resource="&expr;#SWRL"/>
  <expr:expressionObject>
        <swrl:AtomList>
          <rdf:first>
            <swrl:IndividualPropertyAtom>
              <swrl:propertyPredicate rdf:resource="#canBeTranslatedTo"/>
              <swrl:argument1 rdf:resource="#InputLanguage"/>
              <swrl:argument2 rdf:resource="#OutputLanguage"/>
            </swrl:IndividualPropertyAtom>
          </rdf:first>
          <rdf:rest rdf:resource="&rdf;#nil"/>
        </swrl:AtomList>
  </expr:expressionObject>
</expr:SWRL-Condition>
```

Figure 2.4: A Process Precondition Expressed With SWRL

grounding Grounding part is what makes an abstract specification to a concrete one by specifying the mapping of inputs and outputs of atomic processes to the actual invokable services. Details regarding protocol and message formats, serialization, transport, and addressing are handled in this part. OWL-S allows for different types of groundings to be used, but the mostly used one is WSDL grounding, which allows any Web service with a WSDL definition to be marked up as a semantic Web service. The mapping is done if the input/output is directly replaceable to WSDL counterparts or by XSLT-

transformations specified by *xsltTransformation* property. An example transformation for the book finder service output are shown in Figure 2.5 and Figure 2.6:

```
<grounding:wsdlOutput>
          <grounding:WsdlOutputMessageMap>
            <grounding:owlsParameter rdf:resource="#Book"/>
            <grounding:wsdlMessagePart>&groundingWSDL;#bookInfo
            </grounding:wsdlMessagePart>
            <grounding:xsltTransformationString>
                        <![CDATA[<xsl:stylesheet version="1.0"
             xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:template match="/">
                <xsl:variable name="X1" select="bookInfo/bookName"/>
                <xsl:variable name="X2" select="bookInfo/isbn"/>
                <xsl:variable name="X3" select="bookInfo/publishYear"/>
                <rdf:RDF xmlns:rdf=
                                "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
                xmlns:concepts="http://localhost:8080/ESODENEME_ATLAS_WEB
                                /owl/Book.owl#">
                <xsl:if test="not(bookInfo='')">
                        <concepts:Book>
                                <concepts:bookName
                                  rdf:datatype="http://www.w3.org/2001/
                                                XMLSchema#string">
                                        <xsl:value-of select="$X1"/>
                                </concepts:bookName>
                                <concepts:ISBN
                                  rdf:datatype="http://www.w3.org/2001/
                                                XMLSchema#string">
                                        <xsl:value-of select="$X2"/>
                                </concepts:ISBN>
                                <concepts:publishYear
                                  rdf:datatype="http://www.w3.org/2001/
                                                XMLSchema#date">
                                        <xsl:value-of select="$X3"/>
                                </concepts:publishYear>
                        </concepts:Book>
                </xsl:if>
                </rdf:RDF>
        </xsl:template>
</xsl:stylesheet>
                          ]]>
                        </grounding:xsltTransformationString>
          </grounding:WsdlOutputMessageMap>
          </grounding:wsdlOutput>
```

Figure 2.5: XSLT Transformation for OWL-S Output Parameter: Book

The OWL-S output paramater *Book* is composed of *bookName*, *ISBN* and *publishYear* properties. The corresponding WSDL complex type elements can be seen in Figure 2.6 for *bookInfo*.

```
<types>
  <xs:schema targetNamespace='http://localhost/bookfinder' version='1.0'
        xmlns:xs='http://www.w3.org/2001/XMLSchema'>
   <xs:complexType name='bookInfo'>
    <xs:sequence>
     <xs:element minOccurs='0' name='bookName' type='xs:string'/>
     <xs:element minOccurs='0' name='isbn' type='xs:string'/>
     <xs:element minOccurs='0' name='publishYear' type='xs:dateTime'/>
    </xs:sequence>
   </xs:complexType>
  </xs:schema>
 </types>
<message name='BookFinder_GetBookInfoResponse'>
  <part name='bookInfo' type='tns:bookInfo'></part>
 </message>
```

Figure 2.6: WSDL Definition of the Output

2.2 Event Calculus Formalism

Event Calculus is a temporal formalism designed to model and reason about scenarios described as a set of events whose occurrences have the effect of starting or terminating the validity of determined properties of the world (called the fluents)[31]. Event Calculus was first proposed by Kowalski and Sergot[32], the one that will be used in this thesis will be the one defined by Shanahan[17]. Event Calculus is based on first-order predicate calculus, and is capable of representing actions with indirect effects, actions with non-deterministic effects, compound actions, concurrent actions, and continuous change[17].

The basic ontology of the event calculus includes actions or events, fluents and time points. A fluent is a proposition whose value is changeable over time. Event Calculus has the following definitions and axioms:

Table 2.1: Event Calculus Ontology and Definitions

Formula	Meaning
Initiates(a, B, t)	Fluent B holds after action a at time t
Terminates(a, B, t)	Fluent B does not hold after action a at time t
Releases(a, B, t)	Fluent B is not subject to the common sense law of inertia after action a at time t
InitiallyP(B)	Fluent B holds from time 0
InitiallyN(B)	Fluent B does not hold from time 0
Happens(a, t1, t2)	Action a starts at time t1 and ends at time t2
HoldsAt(B, t)	Fluent B holds at time t
Clipped(t1, B, t2)	Fluent B is terminated between times t1 and t2
Declipped(t1, B, t2)	Fluent B is initiated between times t1 and t2

Event Calculus axioms are listed below:

$$HoldsAt(f,t) \leftarrow InitiallyP(f) \wedge \neg Clipped(0,f,t) \tag{2.1}$$

$$HoldsAt(f,t3) \leftarrow Happens(a,t1,t2) \wedge Initiates(a,f,t1) \wedge t2 < t3 \wedge \neg Clipped(t1,f,t3) \tag{2.2}$$

$$Clipped(t1,f,t4) \leftrightarrow \exists a,t1,t2\,[Happens(a,t2,t3) \wedge t1 < t2 \wedge t2 < t4 \wedge [Terminates(a,f,t2) \vee Releases(a,f,t2)]] \tag{2.3}$$

$$\neg HoldsAt(f,t) \leftarrow InitiallyN(f) \wedge \neg Declipped(0,f,t) \tag{2.4}$$

$$\neg HoldsAt(f,t) \leftarrow Happens(a,t1,t2) \wedge Terminates(a,f,t1) \wedge t2 < t3 \wedge \neg Declipped(t1,f,t3) \tag{2.5}$$

$$Declipped(t1,f,t4) \leftrightarrow \exists a,t2,t3\,[Happens(a,t2,t3) \wedge t1 < t3 \wedge t2 < t4 \wedge [Initiates(a,f,t2) \vee Releases(a,f,t2)]] \tag{2.6}$$

$$Happens(a,t1,t2) \rightarrow t1 \leq t2 \tag{2.7}$$

The only way the value of a fluent change is by an initiation or a termination action occurrence, as these axioms demonstrate. These axioms provide to query for a fluent's truth value given the initial conditions and the effect expressing *Initiates* and *Terminates* axioms. The following section gives more details about the decision mechanism of planning with Event Calculus.

2.3 Abductive Planning with Event Calculus Planner

Deduction and abduction are two ways of reasoning for plan generation using Event Calculus. Kave Eshghi was the first author who showed the basis of using abduction to solve Event Calculus planning problems[33]. Shanahan has further worked on the problem and analyzed how abduction leads to getting explanations for a planning problem in Event Calculus[34]. Shanahan has written a prolog planner[35] using meta-level clauses expressing the axioms of EC[18]. For instance for EC Axiom 2.2, the following meta-level clause is written:

$demo([holds\_at(F, T3)|Gs1]) :-$
$axiom(initiates(A, F, T1), Gs2),\ axiom(happens(A, T1, T2), Gs3),$
$axiom(before(T2, T3), []),\ demo([not\ clipped(T1, F, T3)]),$
$append(Gs3, Gs2, Gs4),\ append(Gs4, Gs1, Gs5),\ demo(Gs5).$

The solution to the planning problem involves of expressing the domain in Event Calculus definitions stated above and then running the planner to generate the plans. *Initiates*, *Terminates*, *Happens* and other predicates will be expressed as axiom predicates of the form $axiom(a, R)$ where R is a list of prolog literals [b, c, ..., n]. The formula $axiom(a, R)$ holds if there is a clause of the following form:
$a :- b, c, ..., n$

The plan will consist *Happens* formulas and temporal orderings listed as predicates *before(t1, t2)*. Let the goal formula be represented with γ. Given below is the representation of the planning problem in Event Calculus notation where a conjunction $\sum$ of *Initiates*, *Terminates* and *Releases* formula describing the effects of actions, a conjunction δ of *InitiallyP*, *InitiallyN*, *Happens* and temporal ordering formula for describing a narrative of actions and events, and a conjunction Ω of uniqueness-of-names axioms for actions and fluents:

CIRC[$\sum$; Initiates, Terminates, Releases] $\wedge$ CIRC[δ; Happens] $\wedge$ EC $\wedge$ Ω $\models$ γ where EC encapsulates Event Calculus axioms as stated before. Circumscription is applied to the *Initiates*, *Terminates*, *Releases* and *Happens* formulas. Applying circumscription to *Initiates*, *Terminates* and *Releases* formulas causes actions to have no unexpected events. Applying circumscription to *Happens* formulas causes no unexpected event occurrences.

Abductive Event Calculus Planner has the following form of queries:

"abdemo(Goals, Residue)" where "Goals" is a list of atoms (goals). "Residue" is a pair of lists of atoms [RH,RB], where RH contains happens facts and RB contains temporal ordering facts. The residue keeps track of the needed explanations (the happens facts and temporal ordering facts) for abducibles. An abducible is a predicate for which we are trying to find an explanation for. To illustrate the abduction mechanism of the planner, Shanahan gives a simple Blocks World example[34]. Temporal ordering expressions for actions are represented by *E* for the sake of brevity, instead of representing them with *happens(E, T1, T2)* expressions. The domain independent axioms to be used are the following:

$$holds\_at(P,T) \leftarrow happens(E) \wedge E < T \wedge initiates(E,P) \wedge not\ clipped(E,P,T) \tag{2.8}$$

$$clipped(E,P,T) \leftarrow happens(E') \wedge terminates(E',P) \wedge not\ T \leq E' \wedge not\ E' \leq E \tag{2.9}$$

The domain dependent axioms are the followings:

$$initiates(E, on(X,Y)) \leftarrow act(E, move(X,Y)) \tag{2.10}$$

$$initiates(E, clear(Z)) \leftarrow act(E, move(X,Y)) \wedge holds\_at(on(X,Z), time(E)) \wedge Z \neq Y \tag{2.11}$$

$$terminates(E, clear(Y)) \leftarrow act(E, move(X,Y)) \tag{2.12}$$

$$terminates(E, on(X,Z)) \leftarrow act(E, move(X,Y)) \wedge Z \neq Y \tag{2.13}$$

Figure 2.7 shows the search space for query $holds\_at(on(a,x),t0)$. Abduction generates the residue $D' = \{happens(e1), act(e1, move(a,x)), e1 < t0\}$.

A second example shows how the planner tries to use already existing event (*e1*) in the residue but fails at later steps and generates a second branch that uses a newly generated event. This time $t0 < t1 \wedge t1 < t2$ are given and we want an explanation for $holds\_at(on(a,x),t0) \wedge holds\_at(on(a,x),t2) \wedge holds\_at(clear(x),t1)$. Notice that the first goal already generated the search space in 2.7 so, in Figure 2.8 the other portion of the search space is presented for that purpose. This illustrates the use of checking of negated facts in case at some step during abduction, they are found to be provable.

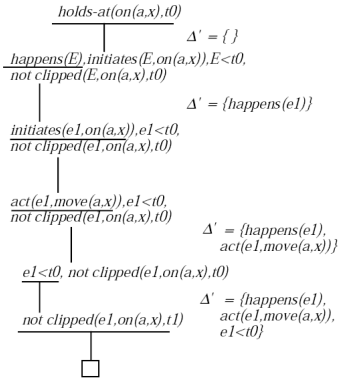


Figure 2.7: Search Space for Blocks World problem - I

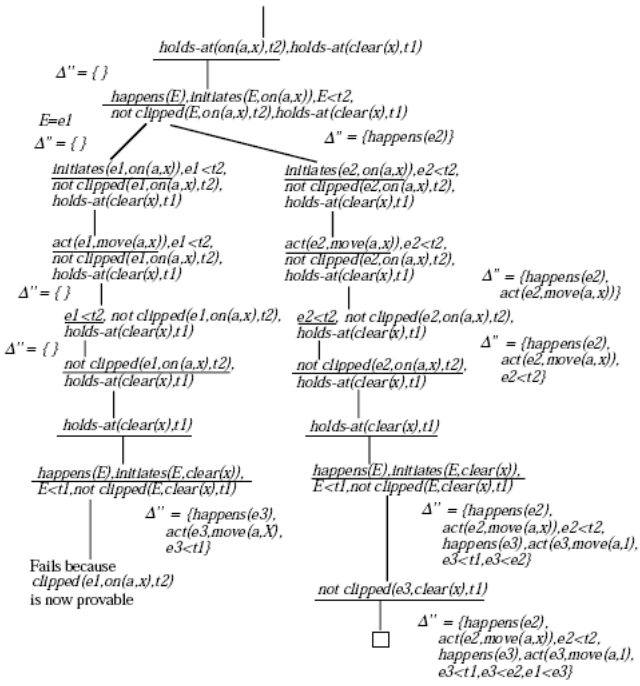


Figure 2.8: Search Space for Blocks World problem - II

2.4 Current Studies on Web Service Composition

The literature presents several approaches to web service composition and execution problem. To clarify the differences between these approaches, [36] proposes four categories for the problem of web service composition and execution (WSCE). This separation is based on the following properties of WSCE characteristics:

- the separability of composition and execution phases
- taking place of composition during design or run-time
- planning or template based composition
- the kind of information used in composition like service types, service instances published/deployed, or service templates

Authors of [36] based on the above properties, stated the following approaches where Table 2.2 shows the relation of these properties with the approaches:

Interleaved Composition and Execution In this approach, the execution takes place during

Table 2.2: Properties of WSCE approaches

Criteria	Interleaved	Monolithic	Staged	Template-based
Separable?	No	Yes	Yes	Yes
When	Online	Offline	Offline	Online/Offline
How	Search	Search	Search	Template
What Information	Deployed Instances	Advertised Instances	Advertised Type & Instances	Templates/ policies Advertised Types & Instances

planning. This way it is possible to dynamically query the environment, rather than to account for all possible world states during planning.

Monolithic Composition and Execution With this approach, specifications are turned into a composite executable workflow which is determined by the planning routine. Run-time aspects of services are not captured, the planning phase only accounts for advertised service properties (IOPE's). After the planning phase several plans may be generated from which *'the best'* solution is selected according to some ranking.

Staged Composition and Execution There are two phases in this approach which are *logical composition* and *physical composition*. This approach distinguishes the service types and instances, which means *logical composition* makes use of service types whereas *physical composition* makes use of actual executable services. Ranking of abstract workflows is also possible with this approach. By introducing an abstract level of composition and service types, this approach makes some kind of filtering on the data set, that is only services of the given service types are considered in the discovery part. Also if there are several services found for replacing a service type in the abstact plan, then a choice decision is made according to some ranking of the actual services.

Template-based Composition and Execution This approach makes use of templates and instantiates them to get an executable workflow. This approach does not compose the workflow automatically as the other approaches do.

The most common web service composition approach in literature is the monolithic approach as will be seen later in this section. Interleaved approach is superior to the others in adapting to the actual environment. With this approach, if a new service is available and/or a service is unavailable the adaptation can be done at the execution time. With other approaches on the other hand, there must be a feedback from the execution to the planning phase. On the other hand, with monolithic or staged composition techniques, it is possible to order the generated

plans according to non-functional requirements or use other ranking techniques. Also with staged and monolithic approaches the failures can be handled at the early stages of the composition by some verification and refinement procedures. In interleaved approach resolving a failure requires a rollback of the executed services. In this thesis, monolithic approach is chosen.

2.4.1 AI Planning Techniques Applied to Web Services Composition Problem

The literature on web service composition using AI planning mainly focuses on the following techniques[37, 38, 39, 40]:

Hierarchical Task Network Planning HTN planning[41] is based on decomposing a task into subtasks which are atomic operations that are executed directly[38].

Situation Calculus Being a formalism for reasoning in dynamic environments, Situation Calculus defines world states with concepts like *actions*, *situations* and *fluents*[39].

PDDL *The Planning Domain Definition Language* [30] is a standard language for most of the AI planners in the literature. It was developed to serve as a standard domain and problem specification language[38]. Since it is used in the initiation of DAML-S, the transformation between them is straightforward.

Rule-based Planning This approach handles composition problem by introducing some composibility rules that are based on syntactic and semantic properties of Web services. The plan is generated given the user specifications for inputs and outputs, and based on the rules defined in the system.

Graph Based Planning This planning technique uses a graph which is expanded to reach all goal states by time steps and then after the graph is created, it searches for a solution which may be a partial order plan as well [42].

Addressing most encountered AI planning techniques applied to web services composition problem, the following section discusses the overview of some of the systems proposed.

2.4.2 Overview of Proposals

Golog is a programming language built on top of Situation Calculus. In [2] the authors propose a software agent, that given a user specified generic workflow and constraints, it is able to execute the workflow with the proper Web services. This agent can be presented as an example of *Template-based Composition and Execution* approach. In this thesis however, a generic workflow is not used, the composition phase generates the workflow.

An example of *Staged Composition and Execution* is IBM's work, named Synthy, presented in [3]. Figure 2.9 shows the general architecture of the system. They have proposed to extend OWL-S ontology by adding an upper ontology which provides *ServiceType* class hierarchy in addition to the *Service* hierarchy. An abstract workflow is generated according to these service types and with physical composition the actual replacements of the services are realized. These replacements are done according to non-functional requirements. In logical composition phase Planner4j is used for planning. The planner is a contingency planner which generates plans according to the the user inputs. The format of the output of physical composition is BPEL which is taken by the execution infrastructure as input. A kind of filtering mechanism is applied to eliminate the service types irrelevant to the goal. The matching of service types with instances is done by using Web Services Matchmaking Engine(WSME) [43] which matches a query record (for each service type in the composed workflow) with an advertisement record (for a web service instance). It is stated that the produced final executable flow needs human review as a last step for checking of data-flow. The idea of using functional requirements for planning phase and non-functional requirements in plan selection phase is similar to the one presented in this thesis. Inputs and outputs are used in instance selection in physical composition whereas this thesis make Abductive Event Calculus Planner use inputs and outputs of services. They are part of the semantic definition of a web service which composibility depends on.

Authors of [4] have proposed a web service composition tool that uses a hybrid planner named Xplan that combines a graph-plan based planner and an HTN planning component. OWL-S service descriptions are transformed to PDDL which are used by Xplan to plan a service composition that satisfies a given goal. The plan is outformatted as a graph. They have developed an event mechanism to simulate the world changing events in order to allow the planner to replan according to the fired events. The events are fired by the user control and

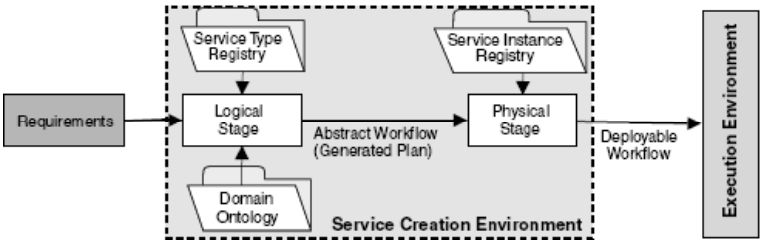


Figure 2.9: Synthy

they are again defined in a file by the user. The output graph needs human interpretation for deciding the plan structure, if it is split/split+join/etc. since the planner only produces ordered sequential plans. In this thesis, however it is also possible to generate partial-order plans. Also the graph is not further processed to make an executable and deployable web service. In our system, the output OWL-S file can directly be executed with user given values for inputs. An earlier version of their tool also uses optimization techniques based on QoS parameters [44]. A similar approach is taken in this thesis also.

[5] is an *Interleaved Composition and Execution* example that uses HTN planning for service composition. They transform OWL-S service descriptions to SHOP2 [45] domain which is an HTN planner that uses domain specific knowledge base of methods to generate a partially ordered set of subtasks decomposition. The architecture offered can be seen in Figure 2.10. Since the planner interleaves planning and execution, only information-gathering services are allowed. Generated SHOP2 plan is converted to an OWL-S service which can be executed. But since SHOP2 does not allow concurrent processes, the output plan cannot contain *split* or *split-join* constructs of OWL-S. Due to the interleaving approach, the preconditions and effects of web services can be integrated to the environment in planning phase providing the knowledge of the current state of the world. In our work, preconditions and effects of web services are both used in planning and execution phase, but during planning evaluation of preconditions does not take place.

[6] proposes a DAML-S virtual machine which is a framework for discovery and execution of web services. Discovery and Execution are part of their infrastructure but they state that planning is the responsibility of individual agents not the infrastructure. They present a web service composition experiment using their framework but there are issues considering lack of support for failure management and recovery of DAML-S in interleaved approach and how preconditions and effects are reflected to planning environment.

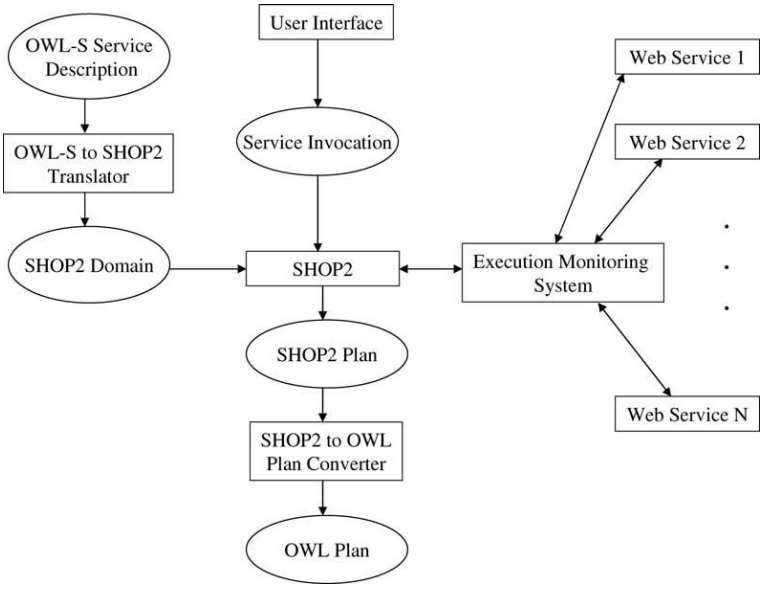


Figure 2.10: SHOP2 Planning Architecture

A *Monolithic Composition and Execution* approach deals with the nondeterministic behavior of Web services, partial observability and complex goals. Authors of [7] propose an automatic web service composition in which OWL-S process models are changed into *State Transition Systems* together with the composition goal. An abstract workflow is generated in their planner's language. This plan is transformed later to an executable BPEL4WS process. Their system supports constructs of sequence, conditions and iterations. Our system takes nondeterministic behavior of web services into account by QoS modelling and in the selection phase. Partial observability and complex goals are not considered in our system.

Another *Monolithic Composition and Execution* approach that uses rule-based planing technique is presented in [8]. They propose ontology-based framework for the automatic composition of Web services. They define composibility rules for comparing syntactic and semantic features of web services to see if they are interoperable. Totally distinct from WSMO or OWL-S, they introduced their Service ontology and composibility rules on the properties of this ontology. WSDL descriptions are annotated with this ontology manually. Then using the defined rules a matchmaking algorithm finds several composite plans from which a plan is selected after applying QoS parameters ranking. By introducing their own ontology they state that they increased the matching of web service capabilities with features like category, purpose, parameters' data types, units, and business roles. The output plan can be converted to WSFL[46], XLANG[47] and BPEL4WS[48]. In this thesis, we use OWL-S for semantically

describing Web services, since it has become widely used standart, which reduces the burden of annotating Web services using another ontology.

[9], proposes a tool for converting semantically annotated WSDL definitions to PDDL domain and uses an appropriate PDDL planner to generate the plan given the user's goal. The planning and execution is again interleaved, for which the failure conditions are delegated to the calling software. An API is presented for expressing complex goals.

2.5 Use of Event Calculus for Semantic Web Service Composition

Event Calculus is used in web service coordination[49, 50], workflow composition based execution[51, 52], workflow generation by planning[1, 53]. With the concurrent actions mechanism, expressivity of action's dependency of preconditions and effects with initiates and terminates axioms and compound actions, Event Calculus framework is very suitable for the solution of web service composition problem. On the other hand, literature mainly have emphasis on use of Event Calculus for web service composibility and verification.

In [49], authors present a framework for verifying properties and detection of inconsistencies both between and within the web services. They propose the use of Event Calculus for discovering two Web services that have no knowledge of one another by using some contract specified as requirements in Event Calculus formalism.

[50] also have a similar approach with [49], Event Calculus is used in providing the contract between the components of composite service and between user and service requirements. They adopted a design-time consistency check beetween components rather than run-time during the composition process. They state that the testing on Event Calculus models are done in early phases of policy development to avoid inconsistency from lack of constraints or too strong constraints. With use of fluents like *permitted*, *obliged* and *invoke* they have modelled contract interactions between services. But automatic web service composition is again out of consideration, they make use of Event Calculus on workflow composition.

A wrapper for Event Calculus is designed in [53] which is called Workflow Specification Language (WSL). The OWL-S semantic web service definitions are transformed together with the user goal to WSL, and by subdividing the goal and using Event Calculus axioms,

workflow is generated. Although the paper gives the architecture and an overview, it lacks details of how Event Calculus axioms are used in resolution while transforming the goals into subgoals.

A workflow based approach is presented in [51], where OWL-S process model is transformed into Event Calculus axioms and generating the user query results in the execution of the plan found by Abductive Event Calculus Planner.

Fully automating the web service composition problem is the main concern in [1], which this thesis is based on and extends the previous work. The approach presented in [1] uses the approach of interleaving planning and execution. OWL-S atomic process models are transformed to Event Calculus axioms. Given the user inputs, outputs and output constraints the execution plan returned by the planner is presented as a graph.

CHAPTER 3

WSCE FRAMEWORK USING THE ABDUCTIVE EVENT CALCULUS PLANNER

In this chapter, general architecture of the proposed WSCE framework is presented. The differences of the framework from the previous work of Kuban[1] is explained in the next section and the proposed framework's workflow steps are discussed in successive sections.

3.1 General Framework

In a previous work[1], an interleaved approach to WSCE problem was introduced and discussed. The purpose of the system was to generate compositions directed by user specified inputs, outputs and output constraints using OWL-S descriptions translated into meta-level axioms to be used with Abductive Event Calculus Planner. The planning domain is restricted to the ontologies described in the registry. Using these ontologies, and inputs and outputs with their values entered by the user, the system is able to present alternative execution paths, as output graphs. Due to the interleaving of planning and execution, all of the presented graphs are actually executed.

Service execution is simulated, there are no calls to actual running services. So OWL-S descriptions are only used for their *process model* part to get IOPEs, *grounding* part is not used.

Figure 3.1 shows two execution plans for the same user query. During the planner searching phase for the abducibles, simulated actual service calls are made and the variables are unified with the actual values returned by these services. This is due to the fact that during

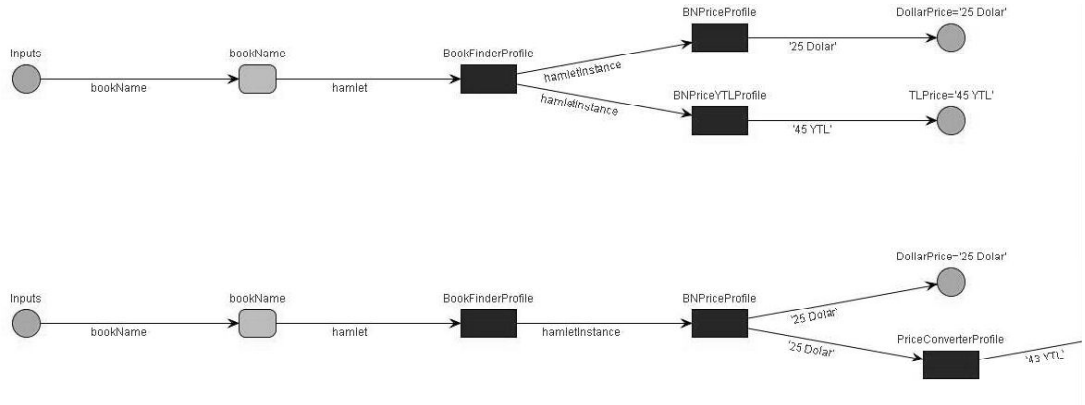


Figure 3.1: Output Execution Plans

transformation of a service to EC axioms, the following kinds of axioms are formed:

```
axiom(happens(pServiceName(Input, Output), T1, TN),
      [jpl_pServiceName(Input, Output) ]).
```

This compound axiom defines a precondition for $pServiceName$ to actually take place, or *happen*. The precondition $jpl\_pServiceName(Input, Output)$ is the call to the actual service where possibly $Output$ variable is bound to some value. Figure 3.2 shows how the actual service call is handled in prolog.

```
jpl_pServiceName(Input, Output) :-
        jpl_new('tr.com.edu.metu.wsc.atlas.main.WebServiceInvocation', [], WSI),
        jpl_list_to_array([Input], InputArray),
        jpl_call(WSI,invokeService,['pServiceName',InputArray],OutputArray),
        statements related to conversion for the output of the Java call to prolog
        true.
```

Figure 3.2: JPL Usage for Actual Service Call

The interface beetween Java and Prolog is handled by JPL which is an API specialized for this purpose.

The work contains also handling of service preconditions and user supplied output constraints. Preconditions that are expressed as SWRL conditions in OWL-S service descriptions, are expressed as another precondition like the service call method:

```
axiom(happens(
      pCalculateNumberOfDays (StartDate, FinishDate, NumberOfDays), T1, TN),
```

```
[
    jpl_pCalculateNumberOfDaysPrecondition('StartDate', 'FinishDate'),
    jpl_pCalculateNumberOfDays(StartDate, FinishDate, NumberOfDays)
])
```

Here *jpl_pCalculateNumberOfDaysPrecondition* call is added before *pCalculateNumberOfDays* action to satisfy before the action actually happens. *jpl_pCalculateNumberOfDaysPrecondition* calls a method by using JPL. The precondition is evaluated before the actual service call and if it fails, then the service call is not executed.

Output constraints are represented in a similar way to preconditions except that the call to constraint check is done after the service call since outputs are needed to be unified before the check. Both preconditions and constraints support the set of SWRL builtins for comparisons.

Populating the actual values to be used for variables by this way yields interleaving of execution and planning. On the other hand this approach has some weaknesses. Direct mapping of complex objects of Java to Prolog is not possible and this may cause problems with composition of services that uses Ontology classes as inputs and outputs rather than simple datatypes. Composition going on-the-fly may prevent user's control on the generated plan. Our method is based on composition of Web services only on their parameter types, not their substituted values. So the calls to actual services in Prolog part has been removed. The output of composition phase is a graph and for the selected plan(s), the OWL-S service description file is generated. The selected service then will be executed with input values taken from the user. This approach is clearly a monolithic approach.

Having an abstract plan before execution gives the user more control on it, like editing if needed and being able to sort the plans according to some ranking algorithm. In this thesis a ranking algorithm based on a set of QoS parameters are used. Details about the usage of QoS parameters are presented in Chapter 4. Although current OWL-S does not specify a standard verification techniques, having the composed plan as an OWL-S file makes it possible for the user to do any verification or refinement.

Because the execution is simulated in [1], the system may not be called a WSCE. In this thesis, we aim to extend the system to a framework from the user specified inputs and outputs to the execution of WSDL-defined web services using the composite process of the output OWL-S file. For this purpose, the output graph generated is further parsed to generate the

control constructs of OWL-S composite process.

This thesis has a different approach in using preconditions. The precondition evaluations cannot take place during the planning phase, since this approach is monolithic. So the preconditions are used in the planning phase for constructing relations between processes whose effect is a precondition to another process. The use of preconditions in this thesis is explained in Chapter 4.

Also this thesis deals with the *subClassOf* property of OWL ontologies, and tries to generate plans taking into account of the class hierarchy.

Web service discovery problem is out of the scope of this thesis. As in [1], a repository of OWL-S services, some of which are adapted from [54], will be used.

Figure3.3 shows the general architecture of the proposed system. The workflow consists of the following steps:

- Converting OWL-S descriptions and user inputs and outputs to Event Calculus axioms
- Converting the Abductive Event Calculus Planner's results to graphs
- Converting the selected graphs to OWL-S service files
- Executing the selected OWL-S service composition.

The following sections give further details of these steps.

3.2 Representing Web Services in EC Domain

The translation of the domain to EC axioms is done similar to [1]. Web services are transformed to executable actions, and service inputs and outputs become action's parameters. A translation for *Book Finder* service, whose *profile* is shown in Figure 3.4 and the translation to prolog is given in Figure 3.5.

Actual datatype properties of inputs and outputs are considered. So *rdf:ID* properties of inputs and outputs are not considered. This requires a global ontology to be used by the services. This service is actually defining its inputs and outputs in terms of *Book.owl* ontology. *BookName* input of the process is actually referencing a datatype in the referenced

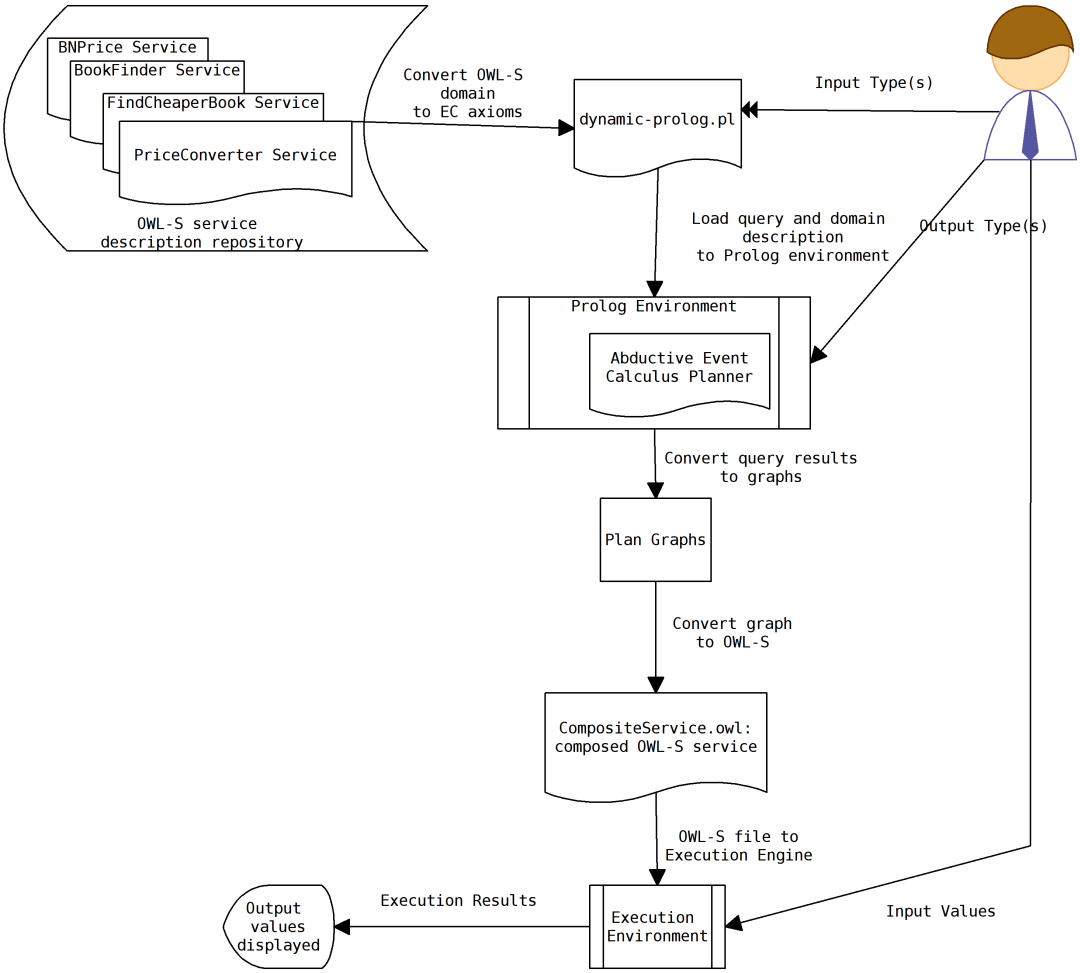


Figure 3.3: Framework

```
<profile:Profile rdf:ID="BookFinderProfile">
        <service:presentedBy rdf:resource="#BookFinderService"/>

        <profile:serviceName xml:lang="en">Book Finder</profile:serviceName>
        <profile:textDescription xml:lang="en">This service returns the
        information of a book whose title best matches the given string.
        </profile:textDescription>

        <profile:hasInput rdf:resource="#BookName"/>

        <profile:hasOutput rdf:resource="#Book"/>

        <profile:hasPrecondition rdf:resource="#AlphanumericCheck"/>

</profile:Profile>
```

Figure 3.4: Book Finder Service Profile

ontology which can be seen in the *parameterType* property of the input which is defined as $\&book;\#bookName$. Figure 3.6 shows how the definitions of inputs and outputs refer to the OWL class and datatype properties of *Book.owl* ontology.

```
executable(pBookFinderProfile(BookName, Book)).

axiom(initiates(pBookFinderProfile(BookName, Book), known(book, Book), T) ,
[
        holds_at(known(bookName, BookName), T), holds_at(alphaNumeric(BookName), T)
]).
```

Figure 3.5: Translation for Book Finder Service

```
...
<process:Input rdf:ID="BookName">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
        &book;#bookName</process:parameterType>

        <rdfs:label>Book Name</rdfs:label>
</process:Input>
..
<process:Output rdf:ID="Book">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
        &book;#Book</process:parameterType>

        <rdfs:label>Book Info</rdfs:label>
</process:Output>
...
```

Figure 3.6: Referring to Book Ontology for Describing Input and Output for Book Finder Service

Book ontology defines the needed datatype and object properties for the ontology classes. A section of the ontology related to the *Book Finder* service is shown in Figure 3.7.

```
<owl:Class rdf:ID="Book" />
...
<owl:DatatypeProperty rdf:ID="bookName">
        <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string" />
        <rdfs:domain rdf:resource="#Book" />
</owl:DatatypeProperty>
```

Figure 3.7: A Section from Book Ontology

This way of using semantically annotated parameter types provides the capability of matching inputs and outputs of Web services without considering syntactic property names. *known* is used for the Abductive Event Calculus Planner to find composable services. *known* on inputs and outputs of services can be thought as knowledge preconditions and effects to derive compositions. It's used as an abducible with the help of $holds\_at$ and is put to the residue to find an explanation by the planner. During translation for each input of the service a $holds\_at$

abducible clause is added as a condition. For each output of the service an *initiates* axiom is added as previously shown.

There are actually two ways for an input of a service to be *known*. One is by the help of a service initiating thereby giving an explanation for the abducible, or by initial conditions. Initially the user specified inputs are assumed to be known, which implies that the user will give the necessary values for these inputs at the execution step. The selection of the inputs are made from the parsed ontologies in the repository. The initial conditions are expressed as axioms like the following:

```
axiom( initially(known(bookName, bookName)), [] ).
```

The planner by using the specified initial condition axioms, tries to generate *happens* formulas trying to reach the goals. Goal clauses are derived from the user specified outputs which are chosen from the ontology hierarchy list like the inputs. The goals are expressed as a list of $holds\_at$ formulas having *known* fluents. For example if the user has specified *DollarPrice* and *TLPrice* as outputs then the following query is formed and queried in Prolog environment:

```
abdemo([holds_at(known(tLPrice,TLPrice), t),
        holds_at(known(dollarPrice,DollarPrice), t)],
        R).
```

While the planner is trying to resolve the goals one by one, the first parameter of the *known* fluent takes the role of parameter type, and the second parameter of the *known* fluent takes the role of the variable. So the planner finds composable services according to the parameter types. The variables that are unified are used in the graph generation phase.

The subclass hierarchy of ontologies are converted to axioms like the following:

```
axiom(holds_at(known(price, TLPrice),T),
        [holds_at(known(tLPrice, TLPrice),T)]).
axiom(holds_at(known(price, DollarPrice),T),
        [holds_at(known(dollarPrice, DollarPrice),T)]).
```

This way, we are able to express that if we know *TLPrice* is of type *tLPrice* then we know that it is of type *price* also. The same holds for *DollarPrice* type, too. The planner will use these axioms in order to derive an explanation like $holds\_at(price, Price)$.

3.3 Generating the Output Graph from the Results of the Planner

The planner populates the results in the given residue of the query. Considering the following query:

```
abdemo([holds_at(known(tLPrice,TLPrice), t),
        holds_at(known(dollarPrice,DollarPrice), t)],
        R).
```

R is bound to a list of *happens* and *before* predicates which constitutes a partial order plan for the problem. The first plan generated by the planner for this query is:

```
R = [[happens(pBookNameValidatorProfile(bookName,_),t4),
      happens(pBookFinderProfile(bookName,A),t3),
      happens(pBNPriceProfile(A,DollarPrice),t2),
      happens(pPriceConverterProfile(DollarPrice,TLPrice),t1)],
     [before(t4,t),before(t4,t1),before(t4,t2),before(t4,t3),
      before(t3,t),before(t3,t1),before(t3,t2),before(t2,t),
      before(t2,t1),before(t1,t)]]
```

By using the prolog's backtracking mechanism it is possible to generate other plans as well. The generated plan is further parsed to determine the vertices and the edges of the graph. The temporal orderings and the identical variable names are used to connect the vertices. Figure 3.8 illustrates an example sequential graph generation. The generation of partial order graphs that includes concurrent *happens* predicates are also possible with the temporal orderings found by the planner.

3.4 Generating OWL-S File from the Generated Graph

After the plan is transformed as a *Graph* object, it is converted to an OWL-S file that has a Composite Process formed by parsing the generated plan graph. Generating an OWL-S file from a pre-defined graph has been worked on [55, 5] and converting a UML or UML-based diagrams to OWL-S service descriptions has been worked on [56, 57]. In [55], the OWL-S graph is extracted from a directed acyclic graph. They have presented their algorithm to automatically generate the OWL-S description of the composite service given an input

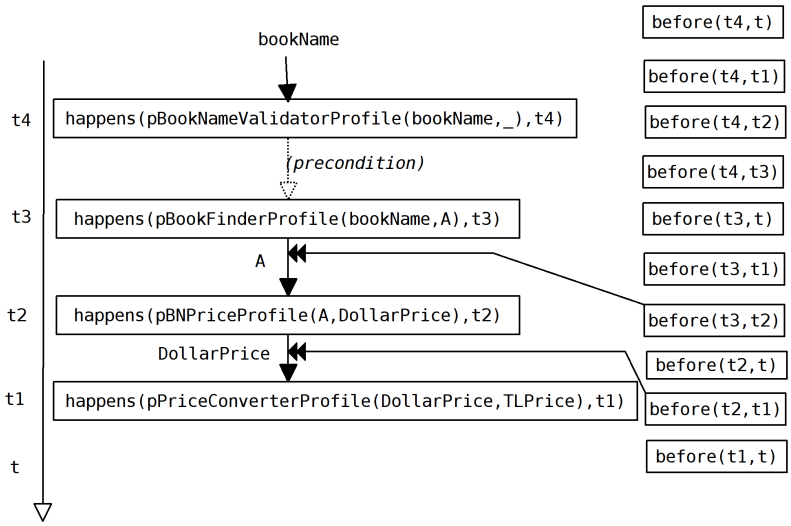


Figure 3.8: Graph Generation

solution graph similar to our case. The algorithm handles the generation of *sequence*, *split-join* and *if-then-else* control constructs. Also in [5], the generated output plan is converted to OWL-S service description. Generated OWL-S composite process cannot include *split* or *split-join* control constructs since the planner does not allow concurrency. Both in [56, 57] a model-driven approach for specifying semantic web service compositions through the use of a UML class and activity diagrams are presented. The aim is automatic construction of OWL-S specifications from UML diagrams. The input to conversion is a user-defined UML diagram, the OWL-S specification is not constructed from scratch hence there is no automatic composition of Web services.

Our conversion algorithm is presented in Figure 3.9 in pseudocode. The algorithm generates composite services having *sequence*, *split* and *split-join* control constructs. This allows solutions for most typical service composition problems including concurrently running atomic process as well. *if-then-else* control construct cannot be handled since the planner only returns a sequence of *happens* predicates and temporal orderings but no information related to conditions that can be used in service composition.

The algorithm comprises both recursion and iteration. Considering a helper vertex at the initial level which has the starting process vertices as children, the algorithm first checks the size of the children. If there is only one child, then a *sequence* construct is created and the flow goes through this vertex by creating a *sequence* construct using *composeProcess*

```
composeProcess(sequence, child, joinVertex):Sequence
{
        if joinVertex is null or child is different from joinVertex
        {
                Create a perform for child vertex
                Add perform to sequence
                composeChildren(sequence, joinVertex, process child list of child)
                return sequence;
        }
        return null;
}
composeChildren(sequence, joinVertex, vertices):void
{
        if(vertices contains only one child)
        {
                composeProcess(sequence, onlyChild, joinVertex);
        }if(vertices contains more than one child)
        {
                joinedChildVertices: map of join vertices to a set of their joined
                                        vertices
                joinedChildVertices = getJoinedChildrenVertices(vertices)

                setOfAllJoinedChildVertices : set of all joined vertices

                joinedSeqs : list of all joined sequence branches

                for each key:vertex in joinedChildVertices
                {
                        Add all joined vertices of vertex to
                                        setOfAllJoinedChildVertices
                        Create new Sequence:seq
                        SplitJoin : splitJoin =
                                        createSplitJoin(joined vertices for vertex,
                                                        vertex)
                        Add splitJoin control construct to seq;
                        Create new Sequence: afterJoinSeq
                        composeProcess(afterJoinSeq, vertex, null);
                        Add afterJoinSeq control construct to seq;
                        Add seq to joinedSeqs;
                }
                if vertices contains any split vertex
                {
                        Create new Split : split
                        for each v in vertices
                        {
                                Add all Sequence constructs in joinedSeqs to split
                                if v is not in setOfAllJoinedChildVertices
                                {
                                        Create new Sequence : seq
                                        Add composeProcess(seq, v, null)
                                                control construct to split
                                }
                        }
                        Add split control construct to sequence
                }
                else if there are no other split vertex
                {
                        Add all Sequence constructs in joinedSeqs to sequence
                }
        }
}
```

Figure 3.9: The Algorithm for Constructing OWL-S Composite Service From Generated Graph

```
createSplitJoin(joinedVertices, joinVertex):SplitJoin
{
        Create new SplitJoin : splitJoin
        for each vertex in joinedVertices
        {
                Create new Sequence : sequence
                Add composeProcess(sequence, vertex, joinVertex)
                        control construct to splitJoin
        }
        return splitJoin;
}

getJoinedChildrenVertices(vertices):Map
{
        joinedChildrenVertices = map of join vertices to a set of their joined
                                 vertices

        For each tuple v1, v2 that has a join Vertex:v
        {
                Add (v, [v1, v2]) to joinedChildrenVertices
        }
        return joinedChildrenVertices;
}
```

Figure 3.9: The Algorithm for Constructing OWL-S Composite Service From Generated Graph (Cont.)

method. If there are more than one child, then for any children that are combined with a join vertex in the graph, a map is constructed by giving the joint vertex as key. For each such combined vertices a *split-join* construct is created. These *split-join* constructs are later added to a *sequence* construct combined with the *sequence* created for the part after the join vertex. These and the created *sequence* control constructs for all other (non joint) vertices are added to the newly created *split* control construct. The *joinVertex* argument is populated through the calls, to make sure that the algorithm stops generating the current construct when that vertex is reached, since the subsequent *sequence* control construct after the join vertex is handled where the *split-join* construct is created.

So at the end of the composition process a new service definition is created. The new service definition has inputs and outputs that the user has specified, and has composite process whose atomic processes are readily executable.

3.5 Execution of the Generated OWL-S File

The execution step consists no more than the user specifying values for the service inputs. The results for the outputs and the monitoring of the actual service calls (SOAP messages exchanged, information about service call flow ...) can be seen.

The services used to test the system are modified to make them work properly with the created and deployed web services on the server. This process requires writing XSLT transformation scripts for changing message parts from OWL-S to WSDL and vice versa manually. The output generated on the screen is just what is represented in this transformation. For example let us assume our composite service has a *DollarPrice* output. The result returned from the actual service finding *DollarPrice* is converted to an OWL ontology instance by the XSLT transformation shown in Figure 3.10.

```
<grounding:xsltTransformationString>
<![CDATA[
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
        <xsl:variable name="X1" select="dollarPrice/price"/>
        <xsl:variable name="X2" select="dollarPrice/scale"/>
        <xsl:variable name="X3" select="dollarPrice/currency"/>
        <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:concepts="http://localhost:8080/ESODENEME_ATLAS_WEB/owl/Price.owl#">
        <xsl:if test="not(dollarPrice='')">
                <concepts:DollarPrice>
                        <concepts:amount rdf:datatype="http://www.w3.org/2001/
                                                        XMLSchema#double">
                                <xsl:value-of select="$X1"/>
                        </concepts:amount>
                        <concepts:scale rdf:datatype="http://www.w3.org/2001/
                                                        XMLSchema#int">
                                <xsl:value-of select="$X2"/>
                        </concepts:scale>
                        <concepts:currency rdf:datatype="http://www.w3.org/2001/
                                                                XMLSchema#string">
                                <xsl:value-of select="$X3"/>
                        </concepts:currency>
                </concepts:DollarPrice>
        </xsl:if>
        </rdf:RDF>
</xsl:template>
</xsl:stylesheet>
]]>
        </grounding:xsltTransformationString>
```

Figure 3.10: XSLT transformation for DollarPrice

For example if currency attribute was not provided in the translation, then it would not be possible to see the value of currency in the results. The XSLT transformations may become an important challenge in annotating the current WSDL services to work with Semantic Web domain.

CHAPTER 4

QoS PARAMETERS, PRECONDITIONS AND EFFECTS

The usual input/output/precondition/effect(IOPE)s based on user's requirements are called the functional requirements of a service. Beside the IOPEs, the composition problem also comprises of selecting the *best* services among alternatives. Many services are introduced and used by e-commerce and Web service applications, which provide overlapping or identical functionality. The most appropriate combination is done by choosing from those services according to some non-functional requirements which are distinct for the alternatives, namely quality of service(QoS). With QoS as an optimization technique, it is possible to integrate the dynamic behavior of the Web that comprises of a large and constantly changing number of service providers. There are many studies on analyzing the use of QoS-based composition techniques, which will be discussed in the next section. In the second section of this chapter, the use of QoS for this thesis is explained. The last section introduces the use of preconditions and effects during the planning phase with Abductive Event Calculus Planner, an extension to work in [1].

4.1 QoS-based Composition Techniques

There are different approaches to QoS-Based composition techniques to the Web service composition problem some of which are using linear integer programming, genetic algorithms and local(task-level) selection of services modelling. In [10], the authors propose a QoS Model that is based on atomic task QoS attributes to compute the quality of service for workflows. Their work includes monitoring atomic tasks for re-computing QoS estimates. They have modelled time, cost and reliability aspects. Their algorithm reduces the workflow until one task remains which represents the QoS for the entire workflow. They have integrated their

model to their workflow system. [11] proposes that a QoS modelling should not treat QoS values as deterministic, rather due to the uncertainty of WS characteristics, it should have a random attitude. They present QoS aggregation techniques according to the structure of the workflow as in [10]. Cost, time, availability and reputation aspects of QoS are modelled in [12]. Rather than giving weight to each of these aspects, they have used genetic algorithms to treat the problem as a multiobjective optimization problem. In [13], two proposals to selecting web service according to QoS parameters are presented. One is task-level selection in which the selection is done at the last possible moment without considering the overall composition. The other uses integer programming technique which supports not generating all the possible execution plans to compare their overall scores. [14] defines a QoS ontology using WSMO for annotating service descriptions with QoS. Using their defined quality attributes, which constitutes a rich set such as accuracy, security, reputation, execution time, exception handling, they have proposed their selection model with an optimum normalization algorithm. [15] proposes an extensible QoS model that supports both generic (like execution duration) and business related quality criteria (like transaction). They collect service quality information from both active execution monitoring and user's feedback. [16] proposes two methods for matching service requests of users with services based on a QoS metric. One is using matchmaking, which will be also the base for this thesis and the other is using a genetic algorithm. The first version of OWLS-XPlan[4] provides an option to do post optimization for the generated plan, using a pre-defined set of QoS parameters which are: execution duration, price, reliability and availability. These most common parameters are selected for QoS modelling for this thesis as well. The equivalent services are placed in a file with their rankings for QoS parameters, and using an integer programming technique the best service is replaced in the composition plan.

4.2 QoS-based Modeling in Our Framework

In our framework, the plans generated by the planner, can be sorted according to some aggregation function's return value. The aggregation function and the algorithm is based on the work presented by [16] and [11]. The QoS attributes used for the calculations are execution duration, price, reliability and availability. They are expressed for each service in the repository in a written file. OWL-S standard does not have built-in properties for service quality. Using *serviceParameter* attribute of *profile* it is manually possible to add these parameters.

Most of the literature focus on extending OWL-S ontology to handle the QoS representation problem in a more organized way. On this thesis however the approach of [4] is taken and the parameters and corresponding values are written in a flat file. This is sufficient for the scope of this thesis since we want to show how QoS parameters help in selecting the best composition plan before the execution step.

The attributes and meanings are presented below:

execution duration is the total time of message transformation and request processing.

price is the fee that a service requester has to pay for invoking a web service.

reliability is the probability that a request is correctly responded to for a specified time interval. Reliability is a general measure of service quality and is related to hardware and software configurations of services, the number of failures per day/week/... and the network connections between requesters and providers.

availability is the probability that the service is accessible.

For each solution plan, QoS match score is aggregated from the involving atomic services. A match score is calculated for the composition using matchmaking algorithm presented in [16]. The algorithm is concerned with matching service requests with services based on a range of QoS attributes. In this algorithm each service has a fixed value, S_i, for each attribute *i*, on the other hand each service request has a range, $[RL_i, RU_i]$, for each attribute *i*. Match value for an attribute *i* is calculated by the following formula:

$$MV_i = \begin{cases} 0 & \text{for } RU_i < S_i \text{ or } RL_i > S_i; \\ \omega_i \times \left(\frac{RU_i - S_i}{RU_i - RL_i}\right) & \text{for } RL_i \leq S_i \leq RU_i. \end{cases} \tag{4.1}$$

Here,

RU_i : represents the service request's upper value for attribute *i*;
RL_i : is the service request's lower value for attribute *i*;
S_i : represents the service's value for attribute *i*;
ω_i : is the weight value for attribute *i*.

The aggregated value of the composite plan is calculated based on the work of [11]. For each atomic Web service *a*, four QoS metrics, execution duration, price, reliability and availability are denoted by $D(a)$, $P(a)$, $R(a)$ and $A(a)$ respectively. A composition construct is denoted by *w* and involves several of such atomic web services. The QoS values of *w*, denoted as $D(w)$, $P(w)$, $R(w)$ and $A(w)$ can be computed from those of its constituent atomic web services. Exceptions are not considered in this approach. Below are the definitions of aggregation functions for control constructs: *sequence*, *split* and *split-join*.

sequence: w consisting of a sequence of web services: *(a1; a2; ... ; an)*
execution duration: The execution duration of *w* is the sum of the execution durations of its constituent web services:

$$D(w) = \sum_{i=1}^{n} D(a_i)$$

price: The price of *w* is the sum of the prices of its constituent web services:

$$P(w) = \sum_{i=1}^{n} P(a_i)$$

reliability: The reliability of *w* is the product of the reliabilities of its constituent web services:

$$R(w) = \prod_{i=1}^{n} R(a_i)$$

availability: The availability of *w* is the product of the availabilities of its constituent web services:

$$A(w) = \prod_{i=1}^{n} A(a_i)$$

parallel split/split-join: w consisting of multiple concurrent web services: *(a1; a2; ... ; an)*
execution duration: The execution duration of *w* is the maximum execution duration of its constituent web services:

$$D(w) = Max\{T(a_i)\}$$

price, *reliability* and *availability* are calculated in a similar way to **sequence** control construct.

The overall match score for a matching is the sum of all aggregated match values divided by the number of QoS attributes:

$$MS = \frac{\sum_{i=1}^{n} MV_i}{n} \tag{4.2}$$

The lower and upper bounds of each attribute is specified by the user before the sorting operation. Also weights are assigned to each of the attribute again by the user. When the sorting

operation begins, a match score is calculated using the match values that are calculated aggregating QoS values of services in composition by using Formula 4.2.

4.3 Precondition and Effect Handling

Kuban's work in [1] handles precondition evaluation and does not include handling of events. The service descriptions having preconditions expressed in SWRL are parsed and added as a precondition before the actual service call for the *happens* compound event that is declared for the service description.

In this thesis, preconditions and effects are used in both planning phase and execution phase. Currently, OWL-S does not declare standard process modeling language for specifying Web service preconditions and effects. Therefore the most commonly used SWRL language is used for this purpose. In [5], SHOP2 statements are directly used for expressing preconditions. In [4], PDDL is used for expressing preconditions and effects, through a language they defined as PDDXML to simplify parsing and reading PDDL descriptions. In our case, the preconditions and effects are represented by prolog predicates and SWRL expressions. In the planning phase, the prolog expression is used, while in the execution phase the SWRL expression is used by the execution engine. Figure 4.1 illustrates an example precondition and effect relationship:

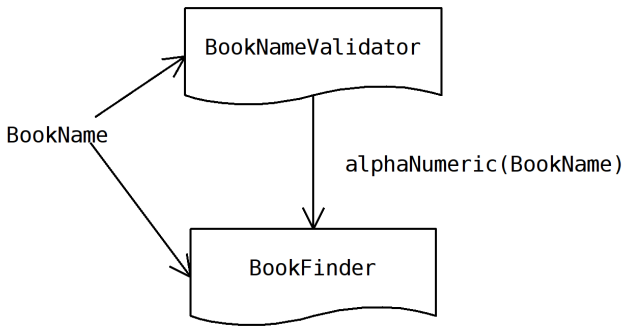


Figure 4.1: Precondition-Effect Relationship Between Sample OWL-S Services

Figure 4.2 shows BookNameValidator service *profile*. The effect of this service is represented by a *Result* property of the *process*. The result depends on the truth value of the service output, namely *ValidationResult*.

```
  <profile:Profile rdf:ID="BookNameValidatorProfile">
    <service:presentedBy rdf:resource="#BookNameValidatorService"/>
    <profile:serviceName xml:lang="en">Book Name Validator</profile:serviceName>
    <profile:textDescription xml:lang="en">This
      service returns true if the given bookName is alphaNumeric
      or contains chars ' ', '_'.</profile:textDescription>
    <profile:hasInput rdf:resource="#BookName"/>
    <profile:hasOutput rdf:resource="#ValidationResult"/>
    <profile:hasResult rdf:resource="#Result"/>
  <profile:hasLocal rdf:resource="#True"/>
  </profile:Profile>
...
  <process:Result rdf:ID="Result">
    <process:inCondition>
      <expr:SWRL-Condition>
        <expr:expressionLanguage
          rdf:resource="http://www.daml.org/services/owl-s/1.1/generic/
                        Expression.owl#SWRL" />
        <expr:expressionObject>
          <swrl:AtomList>
            <rdf:first>
              <swrl:BuiltinAtom>
                <swrl:builtin rdf:resource="&swrlb;#equal"/>
                <swrl:arguments>
                  <rdf:List>
                    <rdf:first rdf:resource="#ValidationResult"/>
                    <rdf:rest>
                      <rdf:List>
                        <rdf:first rdf:resource="#True"/>
                        <rdf:rest rdf:resource="&rdf;#nil"/>
                      </rdf:List>
                    </rdf:rest>
                  </rdf:List>
                </swrl:arguments>
              </swrl:BuiltinAtom>
            </rdf:first>
            <rdf:rest rdf:resource=
                "http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
          </swrl:AtomList>
        </expr:expressionObject>
      </expr:SWRL-Condition>
    </process:inCondition>
    <process:hasEffect rdf:resource="#AlphanumericCheck"/>
  </process:Result>
```

Figure 4.2: BookNameValidator Service Effect Definition

If *ValidationResult* is true then the effect, which is represented by the *hasEffect* property of *Result*, is added to the knowledgebase. This is because *inCondition* property of *Result* is defined as an equivalence between the local variable *True* and the output *ValidationResult*. The modification of the knowledgebase occurs in the execution phase. During planning, the fluent defined in *AlphanumericCheck* is used, which is shown in Figure 4.3.

```
<expr:SWRL-Condition rdf:ID="AlphanumericCheck">
  <rdfs:label>alphaNumeric(BookName)</rdfs:label>
  <expr:expressionLanguage
    rdf:resource="http://www.daml.org/services/owl-s/1.1/generic/
                  Expression.owl#SWRL" />
  <expr:expressionObject>
    <swrl:AtomList>
      <rdf:first>
        <swrl:IndividualPropertyAtom>
          <swrl:propertyPredicate rdf:resource="&book;#alphaNumeric" />
          <swrl:argument1 rdf:resource="#BookName" />
          <swrl:argument2 rdf:resource="&xsd;#boolean" />
        </swrl:IndividualPropertyAtom>
      </rdf:first>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil" />
    </swrl:AtomList>
  </expr:expressionObject>
</expr:SWRL-Condition>
```

Figure 4.3: AlphanumericCheck Condition

The condition has both a textual representation which is represented by the *label* property and a SWRL presentation which is represented by the *expressionObject* property. *label* property is used for defining the prolog predicate to be used in OWL-S service description translations to planner domain. *expressionObject* could also be used by parsing the atom list, but to ease the job and also to have a nicer representation this approach is chosen. The translation of the effect to prolog is done as shown below:

```
axiom(initiates(pBookNameValidatorProfile(BookName, _),
      alphaNumeric(BookName), T) ,
      [holds_at(known(bookName, BookName), T) ]).
```

Figure 4.4 shows how the BookFinder service declares *AlphanumericCheck* as its precondition with *hasPrecondition* property.

```
profile:Profile rdf:ID="BookFinderProfile">
  <service:presentedBy rdf:resource="#BookFinderService"/>

  <profile:serviceName xml:lang="en">Book Finder</profile:serviceName>
  <profile:textDescription xml:lang="en">This service returns the information
         of a book whose title best matches the given string.<
  /profile:textDescription>

  <profile:hasInput rdf:resource="#BookName"/>

  <profile:hasOutput rdf:resource="#Book"/>

  <profile:hasPrecondition rdf:resource="#AlphanumericCheck"/>

</profile:Profile>
```

Figure 4.4: AlphanumericCheck Defined as Precondition to BookFinderService

Translation of the precondition to prolog is done like the following:

```
axiom(initiates(pBookFinderProfile(BookName, Book), known(book, Book), T) ,
        [holds_at(known(bookName, BookName), T),
         holds_at(alphaNumeric(BookName), T) ] ).
```

The precondition is translated like an input for the service, but not using the *known* predicate, rather using the *label* property of the *SWRL-Condition*. Translating an effect of a service is also not much different from an output. In the planning phase we do not know if the effect must actually occur. But having an axiom like this provides service composibility. The planner now is able to generate a plan where BookNameValidator service *happens* before BookFinder service, which at the end turns to a *CompositeProcess* having these two services in a *sequence* control construct.

Thus it is possible to do planing by taking preconditions and effects into consideration. This approach provides a way to use predicates involving inputs of the service for preconditions and involving both inputs and outputs of the service for effects. During the execution phase, the variables are replaced with the actual values. Precondition evaluation and applying effects require reasoning about the state of the world. If a service has a precondition, the knowledgebase is queried for the formed assertion. If a service has a result as a consequence of some condition, the effect is added to the knowledgebase as an assertion. This is provided

by the execution engine used, which is a part of OWL-S API[58] where more information about it will be given in Chapter 5. The execution engine works with an OWL reasoner based on OWL-DL to store the world state, answer the planner's queries regarding the evaluation of preconditions and update the state for the effects of services. More about the use of the reasoner used in execution engine can be found in [59].

CHAPTER 5

IMPLEMENTATION DETAILS AND CASE STUDY

This chapter gives the details of the proposed WSCE with Abductive Event Calculus Planner framework by explaining the used technologies in the first section and a case study which makes it possible to see the mentioned concepts in the implementation.

5.1 Used Technologies and Data Set Preparation

The framework is implemented as a web-based application. Eclipse IDE for Java EE Developers is used for development which provides support for creating Java EE and Web applications. The technology used in client side is JSF (Java Server Faces). JBoss RichFaces component library is used for the user-interface design. SWI-Prolog is used for the prolog execution environment. JPL library is used for providing the connection between prolog execution environment and JVM. For the application server JBoss is chosen. This tool also ease the job of creating running web services. It is only needed to code the web service like coding in a Java Project, and put necessary annotations. OWL-S API is used in a large amount through various phases. The service descriptions in the repository are parsed, the output OWL-S service description is created and executed through this API. Another open source API used in the implementation is Java Universal Network/Graph Framework(JUNG) which is used for generating the graph object of the found plans and also visualization of the plans.

For data set, OWL-S descriptions in [54] are used, and more service descriptions are added for demonstrating the *split* and *split-join* control constructs. Web service implementations are coded in Java as Dynamic Web Project and deployed to JBoss. The implementation bean for

the service is annotated with JAX-WS annotations and deployed to the server. The server automatically generates and publishes necessary contract files like wsdl for client usage. Figure 5.1 is an implementation for BNPrice service:

```
@WebService(name = "BNPrice", targetNamespace = "http://localhost/bnprice")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class BNPrice {

  @WebMethod
  public @WebResult(targetNamespace = "http://localhost/bnprice",
                    name = "dollarPrice")
  DollarPrice getPrice(
      @WebParam(targetNamespace = "http://localhost/bnprice",
                name = "bookInfo")
        BookInfo bookInfo) {
    if (bookInfo.getBookName().equals(
        "Teach Yourself Programming in Ten Years")) {
      System.out.println("bnprice : 23.4");
      return new DollarPrice(23.4, 2, "DollarPrice");
    } else if (bookInfo.getBookName().equals("Object-Oriented Programming")) {
      System.out.println("bnprice : 78.4");
      return new DollarPrice(78.4, 2, "DollarPrice");
    } else {
      System.out.println("bnprice : null");
      return null;
    }
  }
}
```

Figure 5.1: BNPrice Service Implementation

After deployment to the JBoss application server, the wsdl for the service can be seen from the address: `http://localhost:8080/jbossws/services`. The OWL-S grounding for BNPrice is managed accordingly in order to use this service. Chapter 3 shows how this mapping is done. Table 5.1 shows the services in the repository with their IOPEs.

Table 5.1: Services in Repository with IOPEs

Service Name	Input(s)	Output(s)	Precondition(s)	Effect(s)
BNPriceService	Book	DollarPrice		
BNPriceYTLService	Book	TLPrice		
PriceConverter	DollarPrice	TLPrice		
FindCheaperPriceService	DollarPrice, TLPrice	Price		
BookFinderService	BookName	Book	alphaNumeric(BookName)	
BookNameValidatorService	BookName	boolean		alphaNumeric(BookName)

5.2 Case Study

In this case study, it is demonstrated how the WSCE framework is used in an end-to-end service composition problem. The problem is to find the price of a book whose name is known by the user. The application is started by entering the address to a web browser. OWL-S services together with the Ontology definitions are loaded from the repository and ontology hierarchy is presented to the user as a tree structure as shown in Figure 5.2.

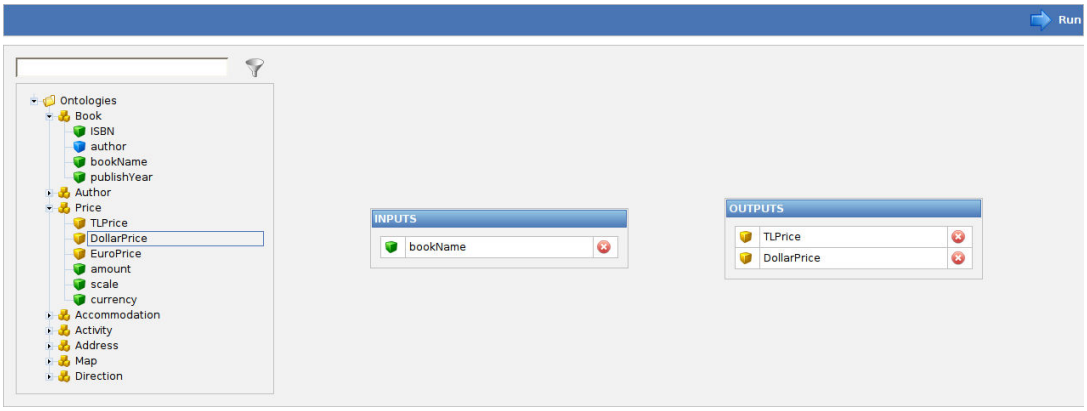


Figure 5.2: Input & Output Selection Phase

On the tree, the ontology classes are shown in yellow, the object properties of classes are shown with blue and datatype properties are shown in green. The subclass hierarchy can also be seen from the tree. The user puts the known knowledge in the *Inputs* section by drag and drop, likewise the requested data is put to *Outputs* section.

When the user clicks on the run button, the translated domain knowledge is written to a file named **dynamic-prolog.pl**. Main axioms added to the domain are listed as follows:

- For each output and/or effect of a web service, an axiom is added for initiating an output to be *known*, or for initiating an effect.
- For each subclass hierarchy of the ontology, an axiom is added to make the planner to search also for subclass type variables while searching for superclass typed variables.
- For each input type given, an initial condition axiom is added to domain knowledge.

Apart from these three groups of axioms, there are facts for the services parsed in order to be

able to add them to the residue as *happens* actions such as the following:

```
executable(pBNPriceProfile(Book, DollarPrice)).
```

For this problem the query for which the planner will find explanations for is:

```
abdemo([holds_at(known(tLPrice,TLPrice), t),
        holds_at(known(dollarPrice,DollarPrice), t)],
        R).
```

This query along with **dynamic-prolog.pl** and the planner are consulted to a prolog session through the call made via JPL. The solutions returned by the planner are parsed to generate the graph object for each plan. For this problem, the following is the first solution generated by the planner.

```
R = [[happens(pBookNameValidatorProfile(bookName,_),t4),
      happens(pBookFinderProfile(bookName,A),t3),
      happens(pBNPriceProfile(A,DollarPrice),t2),
      happens(pPriceConverterProfile(DollarPrice,TLPrice),t1)],
     [before(t4,t),before(t4,t1),before(t4,t2),before(t4,t3),
      before(t3,t),before(t3,t1),before(t3,t2),before(t2,t),
      before(t2,t1),before(t1,t)]]
```

The order of axioms placed in **dynamic-prolog.pl** file also determines the order of the plans returned by the planner. For this example, there are two solutions to get *TLPrice* to be *known*. One is by *PriceConverterProfile* and the other one is by *BNPriceYTLProfile*. While constructing **dynamic-prolog.pl** file, the transformation for *PriceConverter* service is written first, which is the reason the planner generates the first solution using *PriceConverter* service.

The generated graphs are shown with a layout based on Fruchterman-Reingold algorithm whose implementation is provided by the JUNG API. The graphs are shown to the user in the order returned by the planner. Figure 5.3 shows the generated graphs for this example. The graph generation algorithm first adds the vertices to the graph which are inputs, outputs and services. Then using the temporal orderings and the bindings of inputs and outputs of the services, the edges are added to the graph. The edges are formed according to an input-output binding or to a precondition-effect relationship between two services.

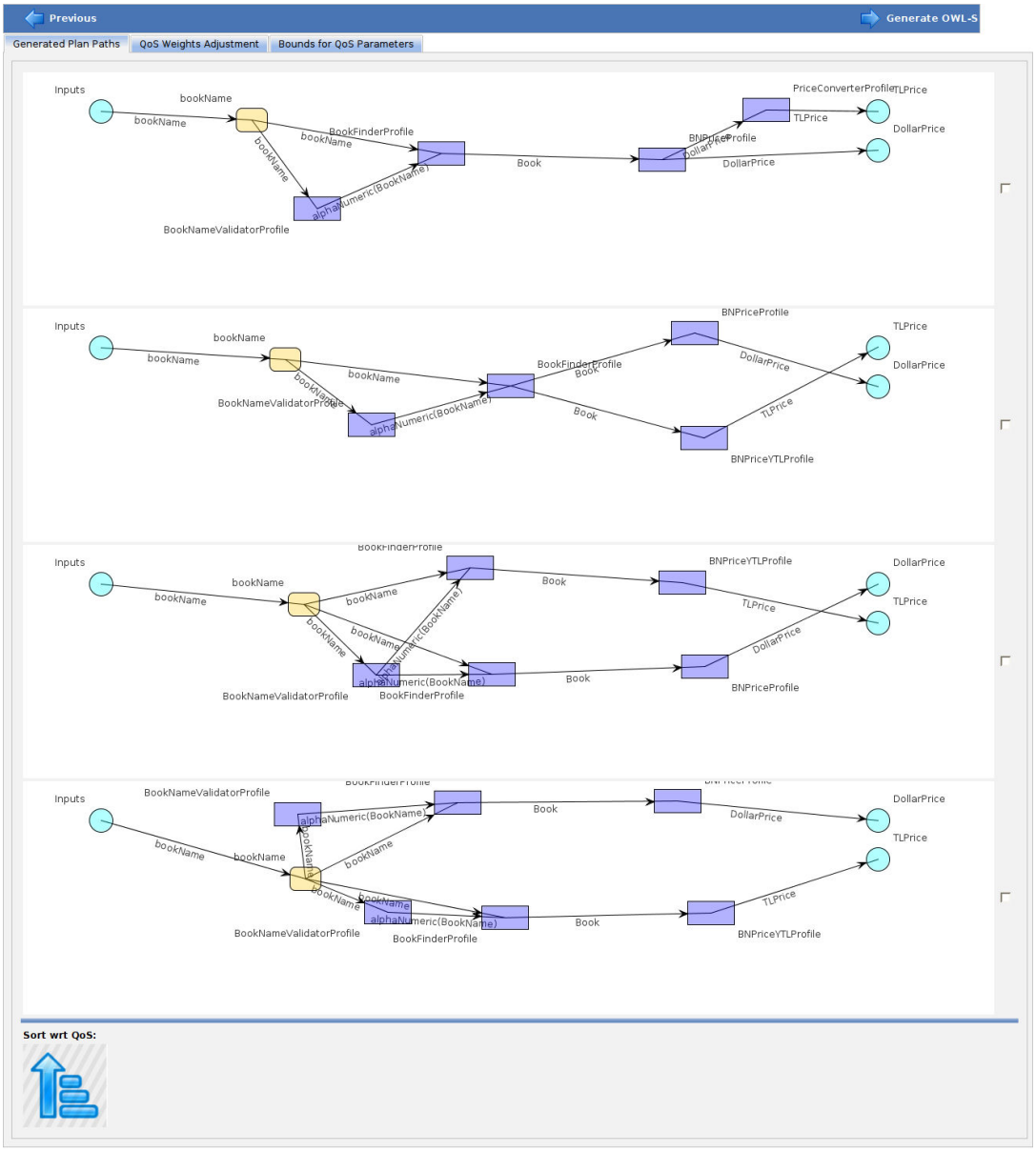


Figure 5.3: Visual Presentations of the Generated Graphs

The user can generate OWL-S descriptions of the composite plans she wants by clicking the boxes near each plan. The system generates OWL-S service descriptions for the graphs selected. The files created are listed on the next page as Figure 5.4 demonstrates.

OWL-S API is used to generate a new service. For each service added to the composition, a *perform* is created by adjusting the inputs and outputs of the atomic process with appropriate bindings. An example *perform* created for process *BNPriceProcess* can be seen in Figure 5.5.

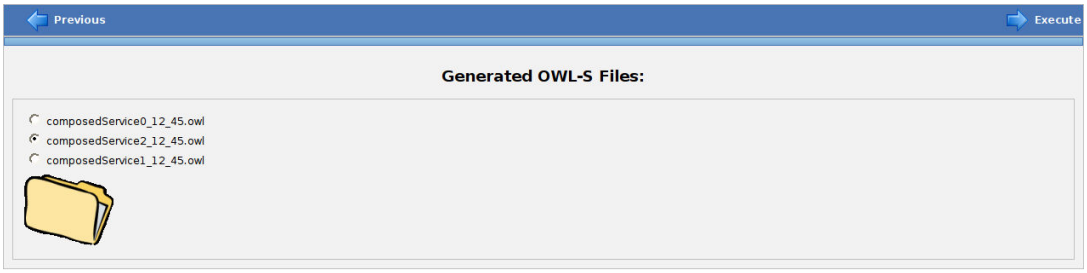


Figure 5.4: Generated OWL-S Files

```
<process:Perform rdf:nodeID="A1">
    <process:process rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/
                                   services/BNPrice.owl#BNPriceProcess"/>
    <process:hasDataFrom>
      <process:InputBinding>
        <process:toParam rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/
                                       services/BNPrice.owl#BookInfo"/>
        <process:valueSource>
          <process:ValueOf>
            <process:fromProcess>
              <process:Perform rdf:nodeID="A0"/>
            </process:fromProcess>
            <process:theVar rdf:resource=
                                    "http://localhost:8080/ESODENEME_ATLAS_WEB/
                                        services/BookFinder.owl#Book"/>
          </process:ValueOf>
        </process:valueSource>
      </process:InputBinding>
    </process:hasDataFrom>
  </process:Perform>
```

Figure 5.5: Created Perform for BNPriceProcess

In OWL-S, a composite process can be considered as a tree whose nonterminal nodes are labeled with control constructs. Control constructs have children specified using components. The leaves of the tree are invocations of other processes that are indicated as instances of class *perform*[26]. The created *perform* for *BNPriceProcess* determines that the invocation of this process needs a binding for its input *BNPrice.owl#BookInfo* from output *BookFinder.owl#Book*. For the first generated graph, a *sequence* control construct is created which includes the *perform*s of the following processes: *BookNameValidatorProcess*, *BookFinderProcess*, *BNPriceProcess* and *PriceConverterProcess*. The full version of the OWL-S file created for the first graph can be seen in Appendix A.

The generated files can be examined with a system editor by clicking the button at the bottom of the screen. The user is able to select the OWL-S file of her choice and execute it by clicking the *Execute* button, which directs to the last page of the WSCE environment from where the execution is performed. Figure 5.6 shows this page. The user enters values for each input

in this screen and later when the execute button is clicked, the composite service starts to run. The *Execution Monitoring Logs* section shows logging information related to service execution after the execution completes. From this section it is possible to see the message exchanges between the services and SOAP messages created during the execution. When the execution is finished, the output values can be seen in the text area near *Outputs* section if the composite service executed successfully.

For the execution of the generated OWL-S file, OWL-S API[58]'s execution engine is used. This engine allows the execution of all OWL-S control constructs. It permits precondition checking before performing processes. Monitoring of the process executions can be done by using the *DefaultProcessMonitor* class or by defining custom monitoring classes. One problem of the API is that the *split* control construct is performed by starting threads in parallel and there is no join of the threads. Each thread executes independently. But this caused some problems when trying to get the output values after executing a composite service that included a *split* construct. This is because OWL-S standard defines *split* control constructs as: "The components of a Split process are a bag of process components to be executed concurrently. Split completes as soon as all of its component processes have been scheduled for execution."[26]. This behavior is changed in our system by joining the threads like *split-join*. Another problem is that it currently does not handle effects. So the execution engine is extended to handle process effects as well. After executing the atomic process, the effects are added to the knowledgebase as properties with actual values replaced for the variables. The effects property can have inputs or outputs as arguments. The effect property is added only if its *inCondition* property holds which is checked by an ABox query created and queried on knowledgebase.

In case the precondition evaluation fails during the execution, it is possible to see the information about the failure of which precondition of what process in the text area on the *Results* section. Figure 5.7 shows such a case. Here the user enters a non-alpha-numeric characters, namely a sequence of '*', which fails the *inCondition* property of *BookNameValidatorProcess* effect. Since the precondition defined in *AlphaNumericCheck* of *BookFinderProcess* is not satisfied, the service is not able to execute. Definitions of the mentioned precondition and effect can be found in Section 4.3.

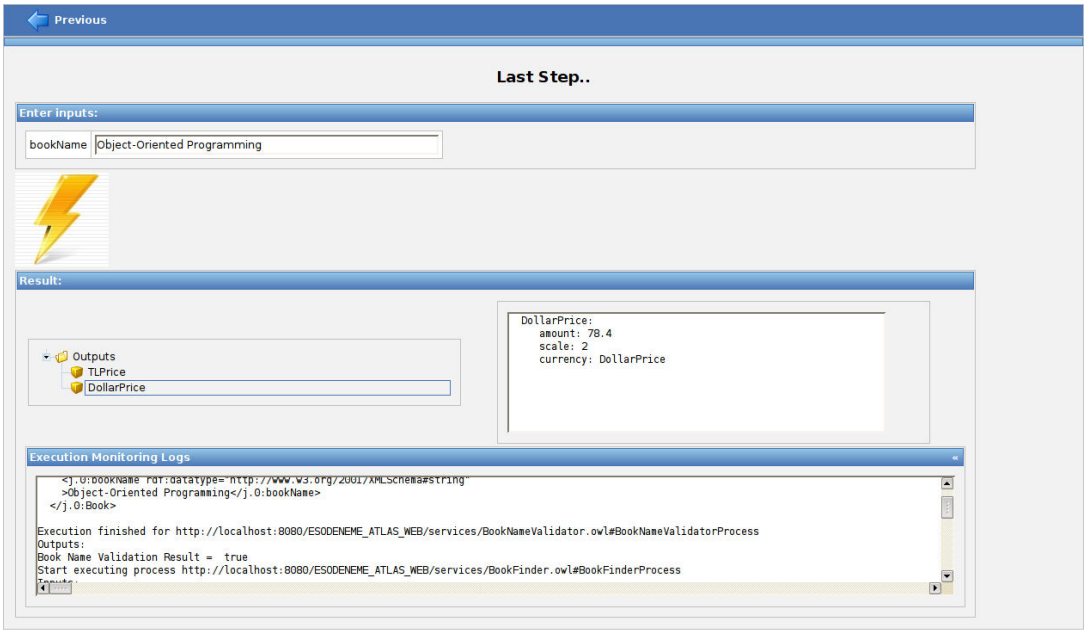


Figure 5.6: Execution Page

5.2.1 Superclass as Output and QoS Sorting

The user can give the requested output as *Price* instead of explicitly giving *DollarPrice* and *TLPrice* as outputs. The query in this case is translated as:

```
abdemo([holds_at(known(price, Price), t)], R).
```

By looking at Table 5.1, we can see that there is only one service which gives a *Price* as output. But the axioms shown in Figure 5.8, which are added to the domain, forces the planner to generate plans to get explanations for making *TLPrice* and *DollarPrice known*. These explanations are the same ones with the found solutions of the previous example. They are generated after the plans containing *happens* actions for *FindCheaperPriceProfile*. This is due to the fact that subclass hierarchy axioms are written to **dynamic-prolog.pl** file lastly. In this case the planner finds more solution graphs each of which has a join node as the last vertex. The solution graphs in this case can be seen in Figure B.1.

QoS parameters become important when there are alternative paths to the given outputs and when some properties of services become important for the user such as cost. In our system, the QoS parameters are used to score the whole composition, but not to replace a service with a better quality in place of another.

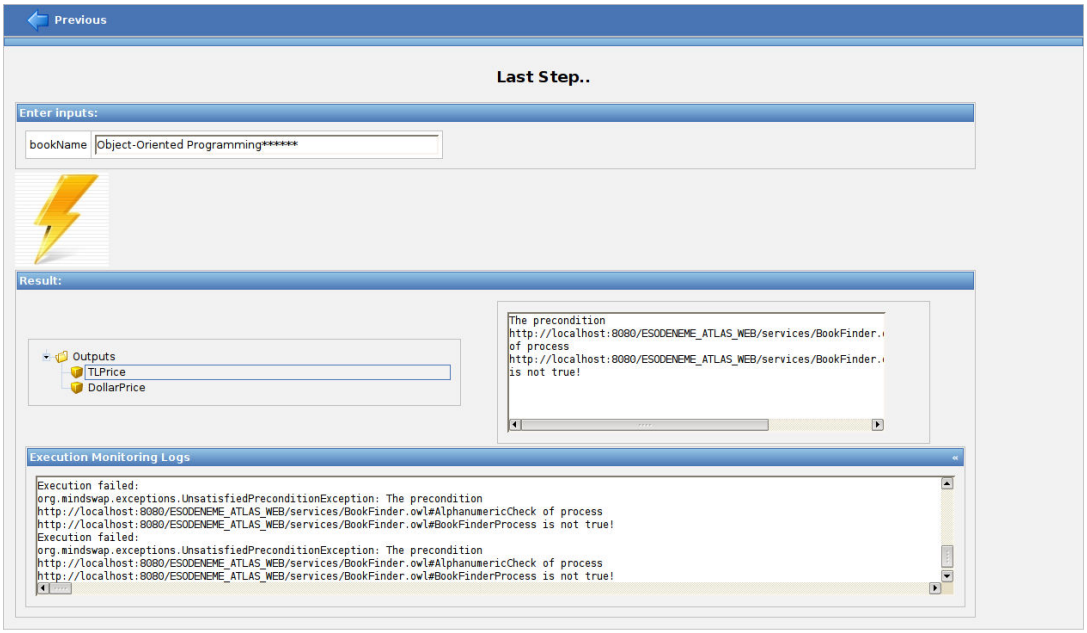


Figure 5.7: Precondition Evaluation Failure

```
axiom(holds_at(known(price, TLPrice),T),
        [holds_at(known(tLPrice, TLPrice),T)]).
axiom(holds_at(known(price, DollarPrice),T),
        [holds_at(known(dollarPrice, DollarPrice),T)]).
```

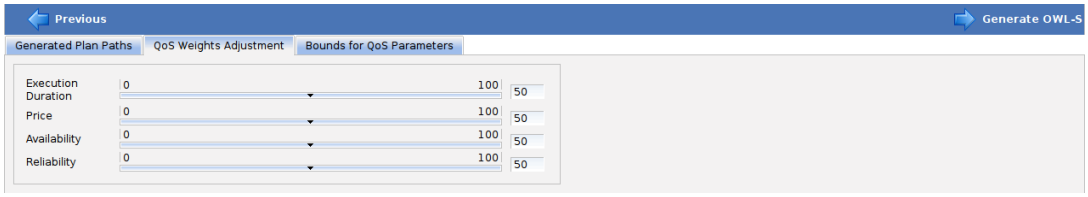
Figure 5.8: Axioms for Deriving *TLPrice* and *DollarPrice* as Outputs

The user clicks the button at the buttom of the screen to sort the solution graphs. The values for QoS parameters are defined in a file. For each service each of the four attributes are given a value. These values correspond to S_i defined in Section 4.2. An example of *BNPriceService* entry is shown below:

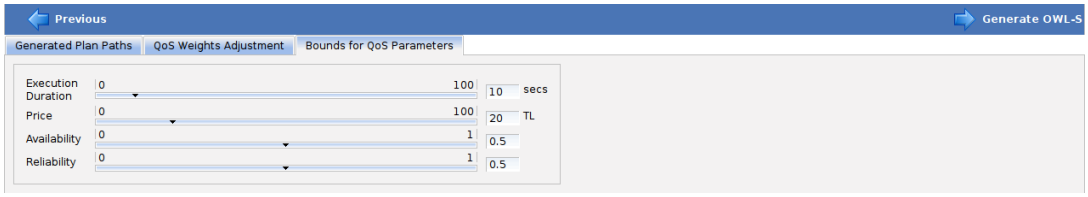
```
BNPriceProfile.execution_duration = 30
BNPriceProfile.price =  20
BNPriceProfile.availability =  0.95
BNPriceProfile.reliability =  0.95
```

The user can adjust the weight of QoS parameters and determine the lower and upper bounds. These actions are shown in Figure 5.9.

A value between 0 and 100 are assigned to attributes, which is divided by 100 to get the percentage weight of the parameter. Assigning weights to each attribute gives the user a chance to determine which attributes are more valueable for them. The percentage of each attribute corresponds to ω_i. The *Bounds for QoS Parameters* section is the area where RU_i



(a) QoS Weights Adjustment



(b) Bounds for QoS Parameters

Figure 5.9: (a):QoS Weights Adjustment, (b):Bounds for QoS Parameters

and RL_i are defined. Here the *execution duration* unit is in seconds, *price* unit is in TL, *reliability* and *availability* are expressed as a percentage. The user defines an upper limit for the *execution duration* and *price*, and a lower limit for *reliability* and *availability*. This means, for example for the *execution duration* that if the upper value is 70 secs, then resulting plans having execution duration more than 70 secs will get a match value of 0 for this attribute. Similarly for *reliability*, if the lower value is 0.7, then resulting plans having reliability less than 0.7 will get a match value of 0 for this attribute. For attributes with lower limit, the difference between the lower limit and service's value must be greater to get a higher match value, similarly for attributes with upper limit, the difference between the upper limit and service's value must be greater to get a higher match value for that attribute.

In Appendix B, Figure B.2 shows the plans after QoS sorting is done. Comparing it with Figure B.1, it can be seen that more complex plans or plans having more count of processes are put towards the buttom of the list.

As a second example demonstrating the effect of QoS based sorting, a new alternative service to *BNPriceYTLService*, given name *YTLBookPriceFinderService* is added to the domain. The QoS values for each attribute of the services are given below:

```
BNPriceYTLProfile.execution_duration = 3
BNPriceYTLProfile.price =  3
BNPriceYTLProfile.availability =  0.8
BNPriceYTLProfile.reliability =  0.8

YTLBookPriceFinderProfile.execution_duration = 10
```

```
YTLBookPriceFinderProfile.price = 4
YTLBookPriceFinderProfile.availability =  0.9
YTLBookPriceFinderProfile.reliability =  0.9
```

The input is *bookName* and the output is *TLPrice* for this example. One of the of the plans generated by the planner involves *BNPriceYTLService* and another involves *YTLBookPriceFinderService*, with the identical execution flow and services other than these services. Figure 5.10 shows the generated plans.

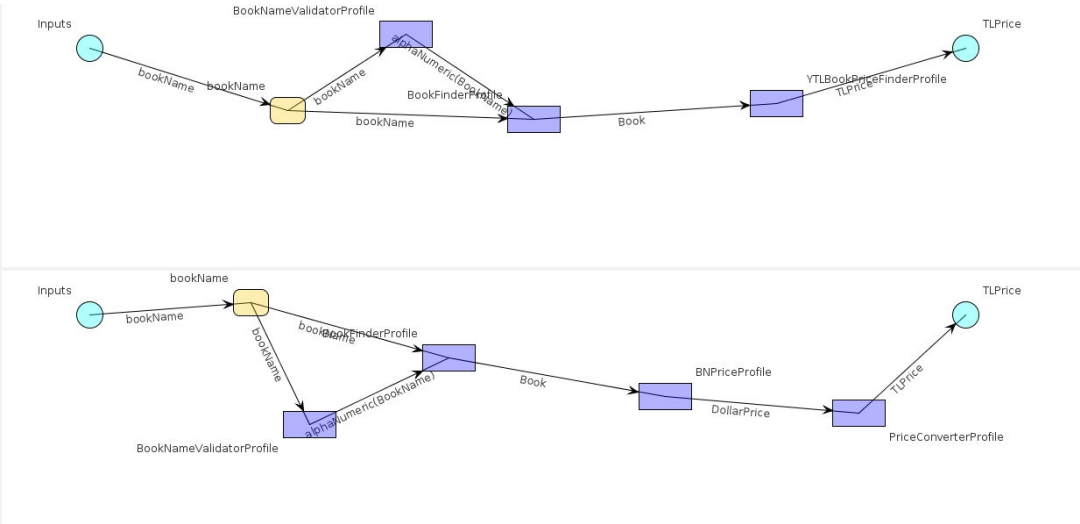


Figure 5.10: Alternative Plans Differing with Only One Service

With these values and the controller values given in Figure 5.9, a match score higher than the plan including *YTLBookPriceFinderService* is calculated for the plan including *BNPriceYTLService*.

```
Match Score for plan involving BNPriceYTLService = 0.19225
Match Score for plan involving YTLBookPriceFinderService = 0.181125
```

If we adjust the weights as giving value 35 for *execution duration* and *price*, giving 80 for *availability* and *reliability*, then the calculated match score for the second plan becomes higher than the first one, since *availability* and *reliability* values for *YTLBookPriceFinderService* is higher than *BNPriceYTLService*:

```
Match Score for generated plan involving BNPriceYTLService = 0.251605
Match Score for generated plan involving YTLBookPriceFinderService = 0.2673
```

CHAPTER 6

CONCLUSION

In this thesis, we propose a Web service composition and execution framework in which the use of Event Calculus formalism is demonstrated to solve the automated Web service composition problem. The system adapts a monolithic approach in which there are two phases namely the composition phase and the execution phase. The composition phase uses the advertised functionalities of web services but does not take into account the deployment and runtime aspects of services. The composition phase may produce more than one composition plan which can be sorted and edited before it is passed to the execution environment.

In the composition phase, the service descriptions which are in the repository and expressed in OWL-S, are transformed to axioms in Event Calculus domain. The Abductive Event Calculus Planner can then generate the plans given a goal state, using this domain information in Prolog language. Event calculus is a logical formalism which allows to reason about the values of *fluent*s by the actions and their effects. With Event Calculus it is possible to express which fluents hold at what time, which actions occur at what time and the effects of actions. This formalism is suitable for Web service composition problem as well, by substituting web services with actions and Web service inputs/outputs with action's knowledge preconditions/knowledge effects and web service preconditions and effects by action preconditions and effects. The plans generated may be ordered or partially ordered according to the given temporal orderings of the actions in the solution plan. OWL-S *process model* is used for extracting the semantic information for Web services. The composibility between web services depend on the *parameterType* property between their inputs and outputs. The *parameterType* can either be a property or an OWL class defined in ontology files. The inputs provided by the user are encoded as initial condition axioms and the outputs defined are encoded as goals. The results of the composition phase are presented to the user as visual

graphs. These graphs can be sorted according to their defined QoS parameters by using an aggregation function on atomic services of the composition plan.

In the execution phase, the selected composition graph is transformed to a service description in OWL-S and passed to the execution engine. The user gives the actual values of the inputs to actually run the atomic services. Transforming to OWL-S includes the steps for determining the control flow and data binding by analyzing the graph. Because the planner allows concurrent actions, the OWL-S composite service may be composed of *sequence*, *split* and/or *split-join* control constructs. For the data bindings, the labels on edges of the graph are used. The *grounding* parts of the services in the repository define XSLT transformation scripts to exchange data between OWL-S and WSDL parameters for inputs and outputs. Preconditions and effects are used in defining the control flow in the planning phase. In the execution phase actual precondition evaluation and effects applying take place which may fail with respect to the current state of the world.

The *execution duration*, *price*, *reliability* and *availability* are the chosen QoS parameters for ranking plans, which are expressed in a configuration file for each service in the repository. Using these attribute values and the user assigned weights and boundary values for each attribute, a match score is derived for the composition plan according to some formula. The user is able to sort the composed plans according to the calculated aggregated value and choose an optimal solution.

This thesis demonstrates that Event Calculus can be used in generating composition plans for the automated web service composition problem by the implementation of a WSCE framework. The WSCE framework tool, implemented as a proof of concept, is the first tool for automating the process of Web service composition and executing the composed service, that makes use of Event Calculus in planning phase. It is shown that the abductive planning techniques that encapsulates Event Calculus and the domain knowledge transformed from OWL-S *process model*s can be used to generate composition plans. The dynamic behavior of the Web is integrated to the system easily by using the representational capability of Event Calculus formalism. This formalism allows modeling of concurrency and temporal ordering between Web service calls which gives our tool the ability to generate plans in the planning phase to find solutions to real world problems including concurrent actions.

A monolithic approach to the problem of WSCE is chosen. This brings more resilience to

failure since it is possible to select an optimal plan after the composition phase. The QoS parameter values become crucial to select the most appropriate services for that purpose. Also with this approach the user is given more control on the workflow before it is passed to the execution environment, which may further help solving inconsistencies before the execution step.

Web service composition problem is a hot research topic. It holds many challenges like non-deterministic behavior of web services, scalability and service side effects that cause change in the world state. Current OWL-S standards do not define a model for error handling and reporting. In this thesis the output workflow is described as an OWL-S service. This may cause low failure resolution which can be compensated by selecting an optimized plan with an adjustment of service QoS parameters like *reliability* and *availability*. The challenges in monitoring executions and dealing with errors are mentioned comprehensively in [60]. In a future work, the execution engine may be extended to handle these problems.

In this thesis, there is no web service discovery module. A discovery module would make the system more realistic, where there is a large collection of web services and some techniques of matchmaking must take place before the problem is handled as a planning problem. This module may also extend the system to a two-staged approach as in [3], where there is a logical composition phase in which the service types are used in composition instead of actual services. In the physical composition phase, the service types are replaced with actual services according to some ranking function.

As mentioned in Section 4.1, there are many algorithms in QoS-based composition techniques to Web service composition problem. In this thesis, we took a rather simple approach for QoS aggregation to select an optimal solution plan for illustrative purposes but there are enormous efforts also in QoS-based composition techniques. Some other algorithm may be replaced with the current one for the problem of selecting the optimal solution plan as a future work.

REFERENCES

[1] E. K. Kuban, "Abductive planning approach for automated web service composition using only user specified inputs and outputs." M.Sc. Thesis in Computer Engineering Department of Middle East Technical University, 2009.

[2] S. Mcilraith and T. C. Son, "Adapting golog for composition of semantic web services," in *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning*, 2002.

[3] V. Agarwal, K. Dasgupta, N. Karnik, A. Kumar, A. Kundu, S. Mittal, and B. Srivastava, "A service creation environment based on end to end composition of web services," in *WWW '05: Proceedings of the 14th international conference on World Wide Web*, (New York, NY, USA), pp. 128–137, ACM, 2005.

[4] M. Klusch and A. Gerber, "Semantic web service composition planning with owls-xplan," in *In Proceedings of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web*, pp. 55–62, 2005.

[5] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau, "Htn planning for web service composition using shop2," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, pp. 377–396, October 2004.

[6] K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan, "Automated discovery, interaction and composition of semantic web services," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, pp. 27–46, December 2003.

[7] S. E. Tecnologica, P. Traverso, M. Pistore, I. T. D. Cultura, P. Traverso, and M. Pistore, "Automated composition of semantic web services into executable processes," 2004.

[8] B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid, "Composing web services on the semantic web," *The VLDB Journal*, vol. 12, no. 4, pp. 333–351, 2003.

[9] J. Peer, "A pddl based tool for automatic web service composition," in *Proceedings of Second International Workshop on Principles and Practice of Semantic Web Reasoning, St. Malo, France (6th–10th September 2004)*, vol. 3208 of *LNCS*, REWERSE, 2004.

[10] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut, "Quality of service for workflows and web service processes," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, pp. 281–308, April 2004.

[11] S.-Y. Hwang, H. Wang, J. Tang, and J. Srivastava, "A probabilistic approach to modeling and estimating the qos of web-services-based workflows," *Inf. Sci.*, vol. 177, no. 23, pp. 5484–5503, 2007.

[12] B. B. Claro, P. Albers, and J. K. Hao, "Selecting web services for optimal composition," *ICWS 2005 Workshop*.

[13] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "Qos-aware middleware for web services composition," *Software Engineering, IEEE Transactions on*, vol. 30, no. 5, pp. 311–327, 2004.

[14] X. Wang, T. Vitvar, M. Kerrigan, and I. Toma, "A qos-aware selection model for semantic web services," pp. 390–401, 2006.

[15] Y. Liu, A. H. Ngu, and L. Z. Zeng, "Qos computation and policing in dynamic web service selection," in *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, (New York, NY, USA), pp. 66–73, ACM Press, 2004.

[16] S. A. Ludwig and S. M. S. Reyhani, "Selection algorithm for grid services based on a quality of service metric," in *HPCS '07: Proceedings of the 21st International Symposium on High Performance Computing Systems and Applications*, (Washington, DC, USA), p. 13, IEEE Computer Society, 2007.

[17] M. P. Shanahan, "The event calculus explained," in *Artificial Intelligence Today, Lecture Notes in AI no. 1600* (M. J. Woolridge and M. Veloso, eds.), pp. 409–430, Springer, 1999.

[18] M. Shanahan, "An abductive event calculus planner," *Journal of Logic Programming*, vol. 44, pp. 207–239, 2000.

[19] J. J. Moreau, R. Chinnici, A. Ryman, and S. Weerawarana, "Web services description language (WSDL) version 2.0 part 1: Core language," candidate recommendation, W3C, March 2006.

[20] M. Gudgin, M. Hadley, N. Mendelsohn, J. J. Moreau, and H. F. Nielsen, "SOAP version 1.2 part 1: Messaging framework." http://www.w3.org/TR/soap12-part1/, 14 June 2003.

[21] "Uddi spec technical committee draft 3.0.2," oasis committee draft, 2004.

[22] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, *BPEL4WS, Business Process Execution Language for Web Services Version 1.1*. IBM, 2003.

[23] D. Martin, M. Paolucci, S. Mcilraith, M. Burstein, D. Mcdermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara, "Bringing semantics to web services: The owl-s approach," pp. 26–42, Springer, 2005.

[24] D. Elenius, G. Denker, D. Martin, F. Gilham, J. Khouri, S. Sadaati, and R. Senanayake, "The owl-s editor – a development tool for semantic web services," in *Second European Semantic Web Conference, ESWC 2005, Heraklion, Crete, Greece* (F. Giunchiglia, W. Nejdl, C. Goble, Y. Sure, S. Staab, P. Lord, P. Haase, and C. Gutierrez, eds.), vol. 3532 of *LNCS*, pp. 78–92, Sringer Verlag.

[25] H. Lausen, A. Polleres, and Dumitru, "Web service modeling ontology (wsmo)." available at: http://www.w3.org/Submission/WSMO/, 10 June 2005.

[26] D. Martin, M. Burstein, E. Hobbs, O. Lassila, D. Mcdermott, S. Mcilraith, S. Narayanan, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara, "Owl-s: Semantic markup for web services," tech. rep., November 2004.

[27] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "Swrl: A semantic web rule language combining owl and ruleml," w3c member submission, World Wide Web Consortium, 2004.

[28] G. Klyne and J. J. Carroll, "Resource description framework (rdf): Concepts and abstract syntax," tech. rep., W3C, 2004.

[29] M. R. Genesereth, "Knowledge interchange format: Draft proposed american national standard," *NCITS.T2/98-004*, 1998.

[30] M. Ghallab, E. Nationale, C. Aeronautiques, C. K. Isi, S. Penberthy, D. E. Smith, Y. Sun, and D. Weld, "Pddl - the planning domain definition language," tech. rep., 1998.

[31] L. N. d. Barros and P. E. Santos, "The nature of knowledge in an abductive event calculus planner," in *EKAW '00: Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling and Management*, (London, UK), pp. 328–343, Springer-Verlag, 2000.

[32] R. Kowalski and M. Sergot, "A logic-based calculus of events," *New Gen. Comput.*, vol. 4, no. 1, pp. 67–95, 1986.

[33] K. Eshghi, "Abductive planning with event calculus," in *ICLP/SLP*, pp. 562–579, 1988.

[34] M. Shanahan, "Prediction is deduction but explanation is abduction," in *Proceedings IJCAI 89*, pp. 1055–1060, Morgan Kaufmann, 1989.

[35] M. Shanahan, "Abductive event calculus planners." `http://www-ics.ee.ic.ac.uk/~mpsha/CogRob/planning/planner19a.txt`, 21 June 2009.

[36] V. Agarwal, G. Chafle, S. Mittal, and B. Srivastava, "Understanding approaches for web service composition and execution," in *Compute '08: Proceedings of the 1st Bangalore annual Compute conference*, (New York, NY, USA), pp. 1–8, ACM, 2008.

[37] J. Rao and X. Su, "A survey of automated web service composition methods," pp. 43–54, 2005.

[38] J. Peer, "Web service composition as ai planning - a survey," tech. rep., 2005.

[39] B. J. Chan May KS and B. Luciano, "Survey and comparison of planning techniques for web services composition," tech. rep., 2006.

[40] S. Dustdar and W. Schreiner, "A survey on web services composition," *Int. J. Web Grid Serv.*, vol. 1, no. 1, pp. 1–30, 2005.

[41] K. Erol, J. Hendler, and D. S. Nau, "Semantics for hierarchical task-network planning," tech. rep., 1994.

[42] A. Blum and M. Furst, "Fast planning through planning graph analysis," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pp. 1636–1642, 1995.

[43] C. Facciorusso, S. Field, R. Hauser, Y. Hoffner, R. Humbel, R. Pawlitzek, W. Rjaibi, and C. Siminitz, "A web services matchmaking engine for web services," in *EC-Web*, pp. 37–49, 2003.

[44] M. Klusch and A. Gerber, "Evaluation of service composition planning with owls-xplan," in *WI-IATW '06: Proceedings of the 2006 IEEE/WIC/ACM international conference on Web Intelligence and Intelligent Agent Technology*, (Washington, DC, USA), pp. 117–120, IEEE Computer Society, 2006.

[45] D. Nau, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman, "Shop2: An htn planning system," *Journal of Artificial Intelligence Research*, vol. 20, pp. 379–404, 2003.

[46] F. Leymann, "Web services flow language (wsfl 1.0)." IBM, 18 April 2001. www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.

[47] S. Thatte, "Xlang: Web services for business process design." Microsoft Corporation, 2001. www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.

[48] M. B. Juric, *Business Process Execution Language for Web Services BPEL and BPEL4WS 2nd Edition*. Packt Publishing, 2006.

[49] M. Rouached and C. Godart, "An event based model for web service coordination," in *2nd International Conference on Web Information Systems and Technologies - WEBIST 2006*, (Setúbal/Portugal), 04 2006.

[50] F. Ishikawa, N. Yoshioka, and S. Honiden, "Developing consistent contractual policies in service composition," in *APSCC '07: Proceedings of the The 2nd IEEE Asia-Pacific Service Computing Conference*, (Washington, DC, USA), pp. 527–534, IEEE Computer Society, 2007.

[51] E. Kırcı, "Automatic composition of semantic web services with the abductive event calculus." M.Sc. Thesis in Computer Engineering Department of Middle East Technical University, 2008.

[52] N. K. Cicekli and Y. Yildirim, "Formalizing workflows using the event calculus," in *DEXA '00: Proceedings of the 11th International Conference on Database and Expert Systems Applications*, (London, UK), pp. 222–231, Springer-Verlag, 2000.

[53] L. Chen and X. Yang, "Applying ai planning to semantic web services for workflow generation," in *SKG '05: Proceedings of the First International Conference on Semantics, Knowledge and Grid*, (Washington, DC, USA), p. 65, IEEE Computer Society, 2005.

[54] E. Sirin, "Owl-s services." `http://www.mindswap.org/2004/owl-s/services.shtml`, 20 June 2009.

[55] S. Kona, A. Bansal, M. B. Blake, and G. Gupta, "Generalized semantics-based service composition," in *ICWS '08: Proceedings of the 2008 IEEE International Conference on Web Services*, (Washington, DC, USA), pp. 219–227, IEEE Computer Society, 2008.

[56] G. C. G. John T. E. Timm, "Specifying semantic web service compositions using uml and ocl," in *ICWS 2007 Proceedings*, pp. 521–528, IEEE Computer Society, 2007.

[57] R. Grønmo, M. C. Jaeger, and H. Hoff, "Transformations between UML and OWL-S," in *Model Driven Architecture – Foundations and Applications: First European Conference (ECMDA-FA'05)* (A. Hartman and D. Kreische, eds.), vol. 3748 of *LNCS*, (Nuremberg, Germany), pp. 269–283, Springer Verlag, Nov. 2005.

[58] M. Information and N. D. Lab, "Semantic web agents project (mindswap) owl-s api." http://www.mindswap.org/2004/owl-s/api/, 12 June 2009.

[59] E. Sirin and B. Parsia, "Planning for semantic web services," in *Semantic web services workshop at 3rd international semantic web conference (iswc2004)*, 2004.

[60] R. Vaculin and K. Sycara, "Monitoring execution of owl-s web services," in *Proceedings of OWL-S: Experiences and Directions Workshop, European Semantic Web Conference*, June 2007.

APPENDIX A

A COMPOSED OWL-S SERVICE

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:process="http://www.daml.org/services/owl-s/1.1/Process.owl#"
    xmlns:service="http://www.daml.org/services/owl-s/1.1/Service.owl#"
    xmlns:list="http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl#"
    xmlns="/home/cokutan/ceng/tezz/esat/deneme_workspace_23022009/deneme_workspace
/ESODENEME_ATLAS_WEB/WebContent/generatedServices/composedService0_14_10.owl"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:profile="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
    xmlns:swrl="http://www.w3.org/2003/11/swrl#"
    xmlns:grounding="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
    xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
    xmlns:expression="http://www.daml.org/services/owl-s/1.1/generic/Expression.owl#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xml:base="/home/cokutan/ceng/tezz/esat/deneme_workspace_23022009/deneme_workspace/
ESODENEME_ATLAS_WEB/WebContent/generatedServices/composedService0_14_10.owl">
  <owl:Ontology rdf:about="/home/cokutan/ceng/tezz/esat/deneme_workspace_23022009/
deneme_workspace/ESODENEME_ATLAS_WEB/WebContent/generatedServices/composedService0_14_10.owl">
    <owl:imports rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/owl/Book.owl"/>
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/Process.owl"/>
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/Profile.owl"/>
    <owl:imports rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/services/
PriceConverter.owl"/>
    <owl:imports rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/owl/
Price.owl"/>
    <owl:imports rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/services/
BNPrice.owl"/>
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/Grounding.owl"/>
    <owl:imports rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/services/
BookFinder.owl"/>
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/Service.owl"/>
    <owl:imports rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/services/
BookNameValidator.owl"/>
  </owl:Ontology>
  <service:Service rdf:ID="TestService">
    <service:presents>
      <profile:Profile rdf:ID="TestProfile"/>
    </service:presents>
    <service:supports>
      <grounding:WsdlGrounding rdf:ID="TestGrounding"/>
    </service:supports>
    <service:describedBy>
      <process:CompositeProcess rdf:ID="TestComposite"/>
    </service:describedBy>
  </service:Service>
  <profile:Profile rdf:about="/home/cokutan/ceng/tezz/esat/deneme_workspace_23022009/
deneme_workspace/ESODENEME_ATLAS_WEB/WebContent/generatedServices/
composedService0_14_10.owl#TestProfile">
    <service:presentedBy rdf:resource="/home/cokutan/ceng/tezz/esat/
deneme_workspace_23022009/deneme_workspace/ESODENEME_ATLAS_WEB/WebContent/
```

```
generatedServices/composedService0_14_10.owl#TestService"/>
    <profile:hasInput>
      <process:Input rdf:ID="bookName">
        <rdfs:label>Book Name</rdfs:label>
        <process:parameterType rdf:datatype="http://www.w3.org/2001/
XMLSchema#anyURI"
        >http://localhost:8080/ESODENEME_ATLAS_WEB/owl/Book.owl#bookName
</process:parameterType>
      </process:Input>
    </profile:hasInput>
    <profile:hasOutput>
      <process:Output rdf:ID="DollarPrice">
        <rdfs:label>Book Price</rdfs:label>
        <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
        >http://localhost:8080/ESODENEME_ATLAS_WEB/owl/Price.owl#DollarPrice
</process:parameterType>
      </process:Output>
    </profile:hasOutput>
    <profile:hasOutput>
      <process:Output rdf:ID="TLPrice">
        <rdfs:label>YTL Price</rdfs:label>
        <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
        >http://localhost:8080/ESODENEME_ATLAS_WEB/owl/Price.owl#TLPrice
</process:parameterType>
      </process:Output>
    </profile:hasOutput>
  </profile:Profile>
  <process:CompositeProcess rdf:about="/home/cokutan/ceng/tezz/esat/
deneme_workspace_23022009/deneme_workspace
/ESODENEME_ATLAS_WEB/WebContent/generatedServices/composedService0_14_10.owl#TestComposite">
    <process:composedOf>
      <process:Sequence>
        <process:components>
          <process:ControlConstructList>
            <list:first>
              <process:Perform>
                <process:process rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/services/
BookNameValidator.owl#BookNameValidatorProcess"/>
                <process:hasDataFrom>
                  <process:InputBinding>
                    <process:toParam rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB
/services/BookNameValidator.owl#BookName"/>
                    <process:valueSource>
                      <process:ValueOf>
                        <process:fromProcess rdf:resource="http://www.daml.org/services/
owl-s/1.1/Process.owl#TheParentPerform"/>
                        <process:theVar rdf:resource="/home/cokutan/ceng/tezz/esat/
deneme_workspace_23022009/deneme_workspace/ESODENEME_ATLAS_WEB/WebContent/generatedServices/
composedService0_14_10.owl#bookName"/>
                      </process:ValueOf>
                    </process:valueSource>
                  </process:InputBinding>
                </process:hasDataFrom>
              </process:Perform>
            </list:first>
            <list:rest>
              <process:ControlConstructList>
                <list:first>
                  <process:Perform rdf:nodeID="A0"/>
                </list:first>
                <list:rest>
                  <process:ControlConstructList>
                    <list:first>
                      <process:Perform rdf:nodeID="A1"/>
                    </list:first>
                    <list:rest>
                      <process:ControlConstructList>
                        <list:first>
                          <process:Perform rdf:nodeID="A2"/>
                        </list:first>
```

```
                        <list:rest rdf:resource="http://www.daml.org/services/owl-s/1.1/generic/
ObjectList.owl#nil"/>
                      </process:ControlConstructList>
                    </list:rest>
                  </process:ControlConstructList>
                </list:rest>
              </process:ControlConstructList>
            </list:rest>
          </process:ControlConstructList>
        </process:components>
      </process:Sequence>
    </process:composedOf>
    <process:hasInput rdf:resource="/home/cokutan/ceng/tezz/esat/deneme_workspace_23022009/
deneme_workspace/ESODENEME_ATLAS_WEB/WebContent/generatedServices/composedService0_14_10.owl
#bookName"/>
    <process:hasOutput rdf:resource="/home/cokutan/ceng/tezz/esat/deneme_workspace_23022009/
deneme_workspace/ESODENEME_ATLAS_WEB/WebContent/generatedServices/composedService0_14_10.owl
#TLPrice"/>
    <process:hasOutput rdf:resource="/home/cokutan/ceng/tezz/esat/deneme_workspace_23022009/
deneme_workspace/ESODENEME_ATLAS_WEB/WebContent/generatedServices/composedService0_14_10.owl
#DollarPrice"/>
    <process:hasResult>
      <process:Result>
        <process:withOutput>
          <process:OutputBinding>
            <process:toParam rdf:resource="/home/cokutan/ceng/tezz/esat/deneme_workspace_23022009/
deneme_workspace/ESODENEME_ATLAS_WEB/WebContent/generatedServices/composedService0_14_10.owl
#TLPrice"/>
            <process:valueSource>
              <process:ValueOf>
                <process:fromProcess>
                  <process:Perform rdf:nodeID="A2"/>
                </process:fromProcess>
                <process:theVar rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/services/
PriceConverter.owl#TLPrice"/>
              </process:ValueOf>
            </process:valueSource>
          </process:OutputBinding>
        </process:withOutput>
        <process:withOutput>
          <process:OutputBinding>
            <process:toParam rdf:resource="/home/cokutan/ceng/tezz/esat/deneme_workspace_23022009/
deneme_workspace/ESODENEME_ATLAS_WEB/WebContent/generatedServices/composedService0_14_10.owl
#DollarPrice"/>
            <process:valueSource>
              <process:ValueOf>
                <process:fromProcess>
                  <process:Perform rdf:nodeID="A1"/>
                </process:fromProcess>
                <process:theVar rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/services/
BNPrice.owl#BookPrice"/>
              </process:ValueOf>
            </process:valueSource>
          </process:OutputBinding>
        </process:withOutput>
      </process:Result>
    </process:hasResult>
    <service:describes rdf:resource="/home/cokutan/ceng/tezz/esat/deneme_workspace_23022009/
deneme_workspace/ESODENEME_ATLAS_WEB/WebContent/generatedServices/composedService0_14_10.owl
#TestService"/>
  </process:CompositeProcess>
  <process:Perform rdf:nodeID="A1">
    <process:process rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/services/
BNPrice.owl#BNPriceProcess"/>
    <process:hasDataFrom>
      <process:InputBinding>
        <process:toParam rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/services/
BNPrice.owl#BookInfo"/>
        <process:valueSource>
          <process:ValueOf>
```

```
            <process:fromProcess>
              <process:Perform rdf:nodeID="A0"/>
            </process:fromProcess>
            <process:theVar rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/services/
BookFinder.owl#Book"/>
          </process:ValueOf>
        </process:valueSource>
      </process:InputBinding>
    </process:hasDataFrom>
  </process:Perform>
  <process:Perform rdf:nodeID="A2">
    <process:process rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/services/
PriceConverter.owl#PriceConverterProcess"/>
    <process:hasDataFrom>
      <process:InputBinding>
        <process:toParam rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/services/
PriceConverter.owl#DollarPrice"/>
        <process:valueSource>
          <process:ValueOf>
            <process:fromProcess rdf:nodeID="A1"/>
            <process:theVar rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/services/
BNPrice.owl#BookPrice"/>
          </process:ValueOf>
        </process:valueSource>
      </process:InputBinding>
    </process:hasDataFrom>
  </process:Perform>
  <process:Perform rdf:nodeID="A0">
    <process:process rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/services/
BookFinder.owl#BookFinderProcess"/>
    <process:hasDataFrom>
      <process:InputBinding>
        <process:toParam rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB/services/
BookFinder.owl#BookName"/>
        <process:valueSource>
          <process:ValueOf>
            <process:fromProcess rdf:resource="http://www.daml.org/services/owl-s/1.1/
Process.owl#TheParentPerform"/>
            <process:theVar rdf:resource="/home/cokutan/ceng/tezz/esat/deneme_workspace_23022009/
deneme_workspace/ESODENEME_ATLAS_WEB/WebContent/generatedServices/composedService0_14_10.owl
#bookName"/>
          </process:ValueOf>
        </process:valueSource>
      </process:InputBinding>
    </process:hasDataFrom>
  </process:Perform>
  <grounding:WsdlGrounding rdf:about="/home/cokutan/ceng/tezz/esat/deneme_workspace_23022009/
deneme_workspace/ESODENEME_ATLAS_WEB/WebContent/generatedServices/composedService0_14_10.owl
#TestGrounding">
    <service:supportedBy rdf:resource="/home/cokutan/ceng/tezz/esat/deneme_workspace_23022009/
deneme_workspace/ESODENEME_ATLAS_WEB/WebContent/generatedServices/composedService0_14_10.owl
#TestService"/>
    <grounding:hasAtomicProcessGrounding rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB
/services/BookNameValidator.owl#BookNameValidatorProcessGrounding"/>
    <grounding:hasAtomicProcessGrounding rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB
/services/BookFinder.owl#BookFinderProcessGrounding"/>
    <grounding:hasAtomicProcessGrounding rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB
/services/BNPrice.owl#BNPriceProcessGrounding"/>
    <grounding:hasAtomicProcessGrounding rdf:resource="http://localhost:8080/ESODENEME_ATLAS_WEB
/services/PriceConverter.owl#PriceConverterProcessGrounding"/>
  </grounding:WsdlGrounding>
</rdf:RDF>
```

APPENDIX B

GRAPHS GENERATED FOR PRICE OUTPUT

This section contains two figures that shows the output result graphs before sorting with respect to QoS parameters and after sorting with respect to QoS parameters respectively.

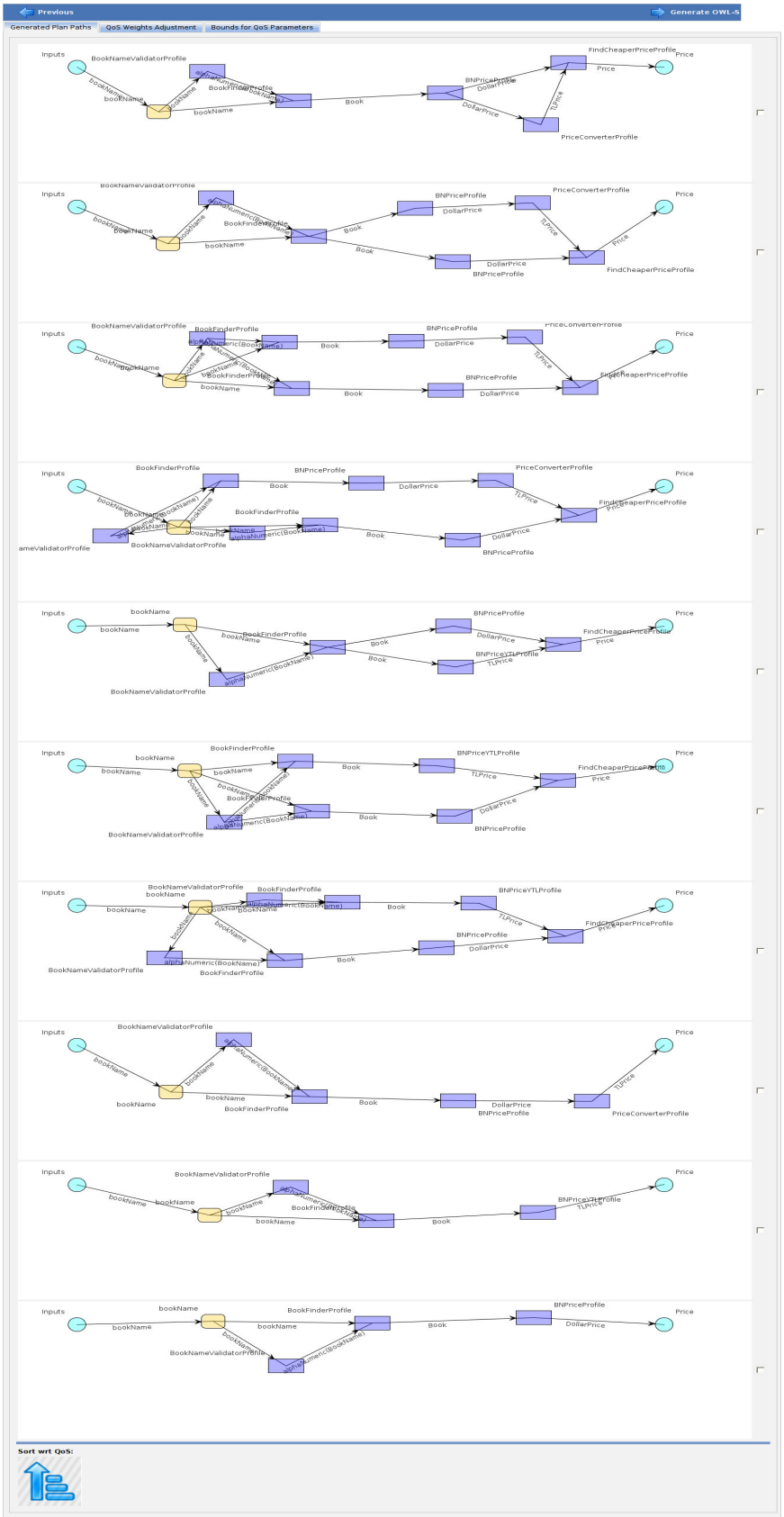


Figure B.1: Graphs before Sorting wrt QoS

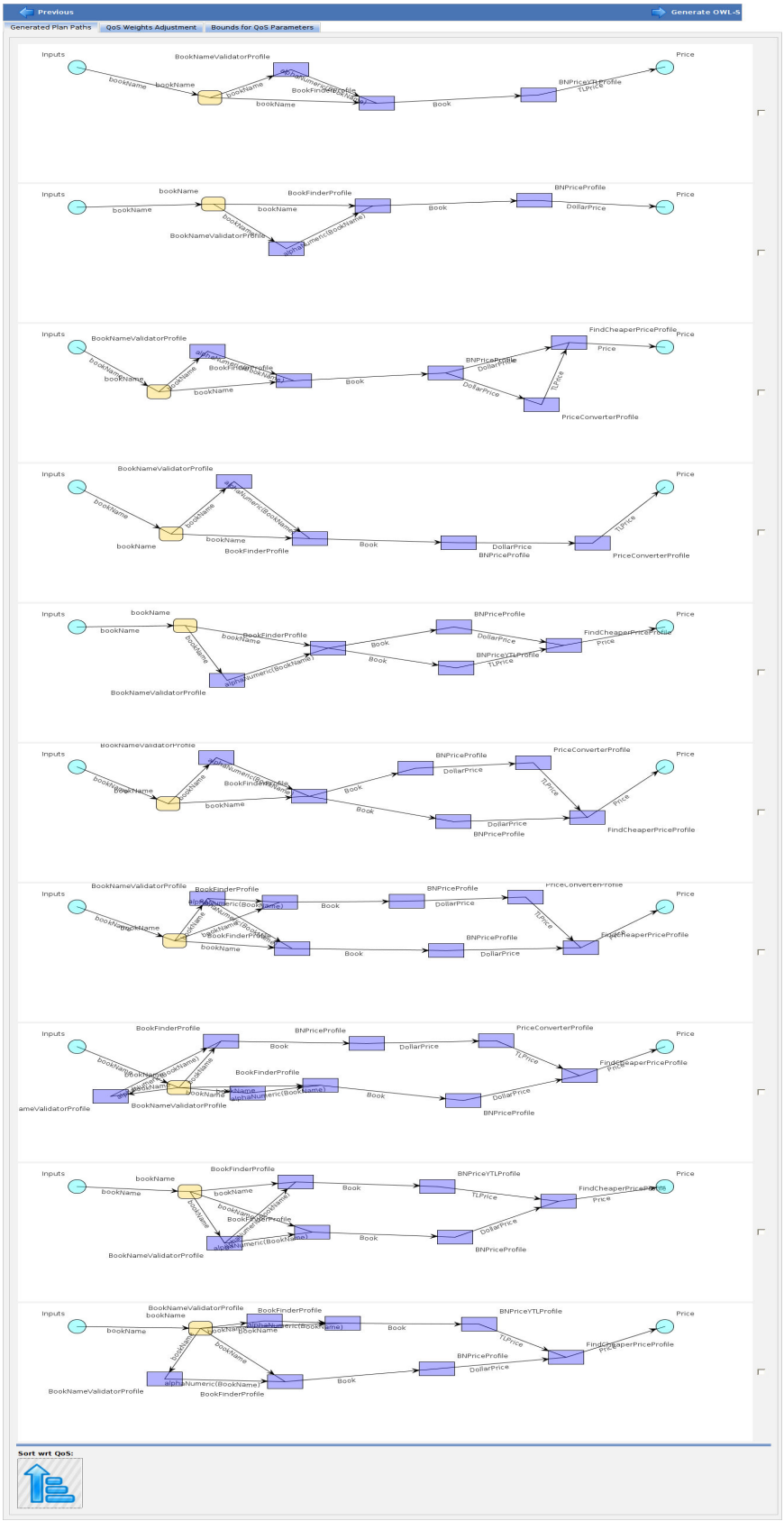


Figure B.2: Graphs after Sorting wrt QoS