ABDUCTIVE PLANNING APPROACH FOR AUTOMATED WEB SERVICE COMPOSITION USING ONLY USER SPECIFIED INPUTS AND OUTPUTS

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

BY

ESAT KAAN KUBAN

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

FEBRUARY 2009

Approval of the thesis:

ABDUCTIVE PLANNING APPROACH FOR AUTOMATED WEB SERVICE COMPOSITION USING ONLY USER SPECIFIED INPUTS AND OUTPUTS

submitted by **ESAT KAAN KUBAN** 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, **Department of Computer Engineering**

Examining Committee Members:

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

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

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

Asst.Prof.Dr. Aysu Betin Can
Information Systems Dept., METU _____

Assoc.Prof.Dr. Ahmet Coşar
Computer Engineering Dept., 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: ESAT KAAN KUBAN

Signature :

ABSTRACT

ABDUCTIVE PLANNING APPROACH FOR AUTOMATED WEB SERVICE COMPOSITION USING ONLY USER SPECIFIED INPUTS AND OUTPUTS

Kuban, Esat Kaan

M.S., Department of Computer Engineering

Supervisor : Assoc. Prof. Dr. Nihan Kesim Çiçekli

February 2009, 76 pages

In recent years, web services have become an emerging technology for communication and integration between applications in many areas such as business to business (B2B) or business to commerce (B2C). In this growing technology, it is hard to compose web services manually because of the increasing number and compexity of web services. Therefore, automation of this composition process has gained a considerable amount of popularity. Automated web service composition can be achieved either by generating the composition plan dynamically using given inputs and outputs, or by locating the correct services if an abstract process model is given. This thesis investigates the former method which is dynamicly generating the composition by using the abductive planning capabilities of the Event Calculus. Event calculus axioms in Prolog language, are generated using the available OWL-S web service descriptions in the service repository, values given to selected inputs from ontologies used by those semantic web services and desired output types selected again from the ontologies. Abductive Theorem Prover which is the AI planner used in this thesis, generates composition plans and execution results according to the generated event calculus axioms. In this thesis, it is shown that abductive event calculus can be used for generating web services composition plans automatically, and returning the results of the generated plans by executing the necessary web

services.

Keywords: Automatic Web Service Composition, OWL-S, Abductive Event Calculus, Semantic Web Services

ÖZ

SADECE KULLANICI TARAFINDAN BELİRTİLEN GİRDİLER VE ÇIKTILAR KULLANILARAK ANLAMSAL ÖRÜN SERVİSLERİNİN OTOMATİK BİRLEŞİMİNE ÇIKARIMSAL PLANLAMA YAKLAŞIMI

Kuban, Esat Kaan
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Doç Dr. Nihan Kesim Çiçekli

Şubat 2009, 76 sayfa

Son yıllarda, örün ağı servisleri İşletmeden İşletmeye ve İşletmeden Tüketiciye gibi birçok alandaki uygulamalar arasında iletişim ve bütünleşme için ortaya çıkan bir teknoloji oldu. Bu büyüyen teknolojide, artan örün ağı servisi sayısı ve karmaşıklığı yüzünden örün ağı servislerinin elle birleşimi oldukça zordur. Dolayısıyla, bu birleşim işleminin otomatikleştirilmesi kayda değer bir popülerlik kazandı. Otomatik örün ağı servis birleşimi, ya birleşim planının verilen girdi ve çıktılar kullanılarak dinamik bir şekilde yaratılmasıyla, ya da özet süreç modeli verildiyse doğru servisleri bularak başarılabilir. Bu tez, olay cebirinin çıkarımsal yetenekleri kullanılarak birleşim planının dinamik oluşturulması olan ilk metodu araştırmaktadır. Prolog dilindeki olay cebiri aksiyomları, servis havuzundaki mevcut OWL-S örün ağı servis tanımları, bu havuzdaki servisler tarafından kullanılan ontolojiler arasından seçilen girdilere verilen değerler ve yine bu ontolojiler arasından seçilmiş istenen çıktı tipleri kullanılarak oluşturulur. Bu tezde kullanılan yapay zeka planlayıcısı olan Çıkarımsal Teorem İspatlayıcı birleşim planlarını ve uygulama sonuçlarını sağlanan bu olay cebiri aksiyomlarına göre oluşturur. Bu tezde, örün ağı servislerinin birleşim planlarının otomatik oluşturulmasında ve oluşturulan planların gerekli örün ağı servisleri icra edilerek sonuçlarının döndürülmesinde çıkarımsal

olay cebirinin kullanılabileceği gösterilmiştir.

Anahtar Kelimeler: Otomatik Örün Servisi Birleşimi, OWL-S, Çıkarımsal Olay Cebiri, Anlamsal Örün Ağları

Dedicated to my parents.

ACKNOWLEDGMENTS

I wish to thank all those who helped me. Without them, I could not have completed this thesis.

I would like to acknowledge Assoc. Prof. Dr. Nihan Kesim Çiçekli who not only supported me as my supervisor but also encouraged and guided me throughout my academic program.

I would like to thank my committee members Assoc.Prof.Dr. Ali Doğru, Asst.Prof.Dr. Pınar Şenkul, Dr. Aysu Betin Can and Assoc.Prof.Dr. Ahmet Coşar for their invaluable comments to improve this thesis.

I especially want to thank my family for their love, support and motivation. I want to thank my friends for their endless supports.

I also want to thank TUBITAK-BIDEB for their financial support.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

Web services are business functions that operate over the Internet via platform and programming language independent interfaces. With the increase in the Internet usage, the number of created and published web services also increase every day. Today most of the companies do business with their partners, and supply customers' needs using web services. This is because of the fact that, integration and interaction among business applications can be done easily using web services. However, in this growing technology, finding the correct services, satisfying user needs and integrating more services in order to serve a purpose of one business are the major difficulties besides their advantages.

Although web services are very generic, there are three specific standards that are widely used as the core of web services technology. The first is the Web Services Description Language (WSDL) [17], a format for specifying the operations that a web service publishes, the transport mechanisms through which the service publishes these operations, and where the service is located. The next is SOAP [35], a protocol that specifies the structure of a message in XML. The last of these is The Universal Description Discovery and Integration (UDDI) [86], a platform free registry standard used to publish and discover services over the Internet. Unfortunately, even with such technologies, the integration of services still depends largely on human experts.

In such a dynamic and flourishing domain, the primary attention is shifted from creating new web services to re-using and composing existing services in order to discover new functionalities. Sometimes, no single web service can satisfy the user requirements. In such situations, more than one web service should be combined and executed to achieve a specified goal. Given a repository of service descriptions and a service request, the web service composition

problem involves finding multiple web services that can be put together in a correct order of execution to obtain the desired service [6]. The composition plan can be modeled manually using static service declarations in the composition plan. This approach, composing services manually, has some drawbacks such that the amount of available web services are too much and they can be changed in both operational and semantic manners or completely deleted on the fly. The systems using manual composition techniques should keep track of used services to be informed the services' current states. In such a dynamic world, it is not possible to achieve this. Thus, automation of web services composition process is unavoidable.

Because of this emerging need for web services composition, there are some languages proposed to describe the composition such as BPML [3], IBM's Web Services Flow Language (WSFL) [37, 48], Microsoft's XLANG [85], and Business Process Execution Language for Web Services (BPEL4WS) [1]. These languages are based on SOAP, WSDL, and UDDI. Web service composition, on the other hand, requires more dynamic functionality and semantic information than SOAP, WSDL, and UDDI can provide. These languages are used in manual service compositions. They do not generate a dynamic composition, but coordinate the created manual composition plan from the point of data and control flow.

In spite of all these approaches, automated web service composition is still a hard problem. One approach in this context that has been proposed to accomplish web service composition is AI planning. In AI planning domain, planning is seen as finding a set of activities to achieve a certain goal. There are *states* in this context, describing the world at a certain point in time and *actions* having effects on the world by changing these states. In AI planning methodology, in order to solve a planning problem; a description of possible actions, a description of the initial state of the world and a description of a desired goal state should be provided. Web services can now be specified in this planning context as actions, inputs and outputs of web services can be regarded as preconditions and effects of actions, user specified inputs and outputs can describe the initial state of the world and the goal state, respectively. And the AI planning can be used to generate a composite service (plan) that achieves the given goal. The automation of web services composition can be done either generating the plan automatically, or finding the correct services if an abstract process model is given [68]. In this thesis we are concerned with creating composition plan automatically without any abstract model is given.

There is a considerable amount of work to solve automated web service composition problem

using AI planning methods [47, 54, 55, 63, 65, 68]. The techniques introduced in these researches are using the situation calculus, the Planning Domain Definition Language (PDDL), rule-based planning, the theorem proving and others. As mentioned in [65], the event calculus [16] is one of the suitable techniques for the automated composition of web services.

In this thesis our aim is to contribute the research along this direction by proposing a formal framework that shows how semantic web services are represented in the event calculus framework to produce user specific composition plans for the requested goals. The web services are described using OWL-S which provides the semantic information about services' process models. To generate composition plans as a result of abductive planning, the semantic web services are translated into event calculus axioms using their input, output, precondition and effect parameters defined in their OWL-S descriptions. In the event calculus framework, instead of parameter names of web services, ontological parameter types facilitated by OWL-S definitions are used in order to prevent the problem of matching the input/output parameters to find web services in a given repository. The outputs desired by the user are modeled as the goal state and given as a query to the abductive planner, which in turn generates the composition plans including necessary web services to achieve the goal and returns the results of those web services.

The rest of the thesis is organized as follows. Chapter 2 reviews related work and gives information about web services, OWL-S, event calculus and the web service composition problem and techniques used to solve the problem. In Chapter 3, we introduce how the event calculus and its abductive implementation can be used as a method for automated web service composition. Also methods to translate OWL-S service descriptions to the event calculus axioms are presented. In chapter 4, we give a description of how preconditions and user defined output constraints are used in our system. The implementation details of the proposed system and a case study are presented in Chapter 5. Finally, Chapter 6 gives conclusions and possible future work.

CHAPTER 2

BACKGROUND & RELATED WORK

2.1 WEB SERVICES

A new paradigm, web services, are introduced to build distributed web applications. The W3C defines web services as software systems designed to support interoperable machine-to-machine interaction over a network [13]. Web services are programmatic interfaces that enable communication among applications. Using these published interfaces, the other applications can communicate with these web services.

Web services are mostly categorized according to the task they are designed to perform. Information-Providing Web Services and World-Altering Web Services are two categories falling into this group. Information-Providing Web Services are the web services that do not alter any state defined in the world and only return information about the current state of the queried content such as weather forecast services, travel information providers, and book information providers. On the other hand, World-Altering Web Services are defined as the services that, when executed, have an effect on their domain [5]. Examples of such services are flight-booking programs and a variety of e-commerce and business-to-business services.

The main reason behind the fact that web services are the key technology in application communication and integration is that, web services do not depend on the running platform and the implementation language. In order to provide these independencies, some standards are defined on web services. Three technologies that roughly correspond to HTML, HTTP, and URIs in 3W architecture are core to defining Web Services [25]:

1. WSDL (Web Services Description Language) is a language based on XML that is a

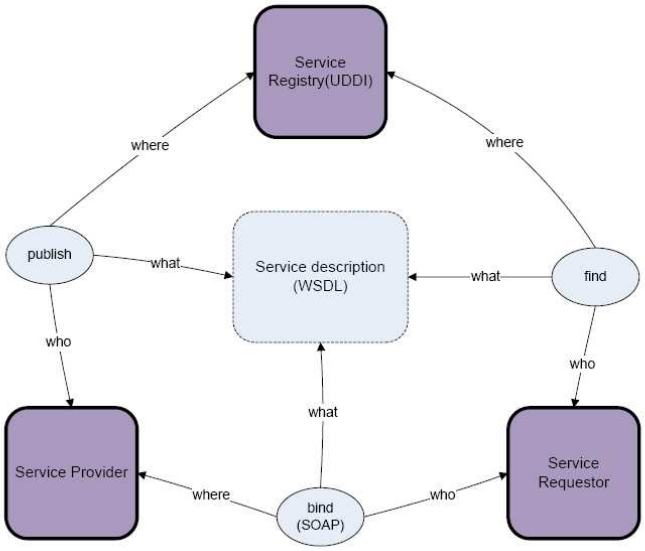

Figure 2.1: Web Service Framework.

standardized way to describe service structures such as operations, messages types, bindings to protocols, and endpoints [17]. In the context of WSDL, web services are regarded as software components that encapsulate and provide a set of closely related operations associated with a set of resources.

2. SOAP (Simple Object Access Protocol) [35] is a protocol specification that defines a uniform way of exchanging XML data in the implementation of Web Services.

3. UDDI (Universal Description, Discovery and Integration) [86] is a platform independent registry that allows the software developers to discover available Web Services that are listed.

There are three different kinds of agents shown in Figure 2.1, namely service provider, service requestor, and service registry. Between the agents, three kinds of interactions are defined. “Find” is a service discovery which is performed by a service requestor. “Publish” is the operation of storing a service description into a registry agent, performed by a service provider. “Bind” is the name of the step used by service requestor in order to connect to a web service at a particular web location (endpoint) and start interacting with the service [73].

By the end of 2006, the public UDDI registries, which were considered as the global solution for service discovery, were shut down blowing the idea of UDDI. The main reasons for that

are; the need for keywords, service name and manual selection of discovered services, lack of coverage of the web services available publicly and the simplicity of the available search tools [7].

2.2 OWL-S

According to W3C, OWL (Web Ontology Language) is a semantic markup language for publishing and sharing ontologies on the World Wide Web [9]. OWL-S (formerly DAML-S) is an ontology in OWL for describing web services semantically. It has been developed within the DARPA/DAML program and currently is a W3C recommendation. So OWL-S is an attempt to combine the representational technologies of the Semantic Web such as RDF and OWL with the dominant Web services standards, such as WSDL.

OWL-S consists of the following ontologies [57]: The topmost level consists of a Service ontology. The Service is described in terms of a ServiceProfile, ServiceModel and a ServiceGrounding ontology, which are as follows:

1. The service presents a ServiceProfile which has a subclass Profile. The Profile specifies what a service does. It is used to enable advertising, construction of service requests, and matchmaking. Profile contains a representation to characterize properties of the service provider, functional properties of the service like Inputs, Outputs, Effects and Preconditions (IOPEs) and non-functional properties of the service. After the selection and engagement of service, the profile becomes useless; the process model will be the basis of interaction among the requester and provider.

2. The service is described by a ServiceModel which has a subclass called Process. The Process model describes how a service works. The Process consists of all the functional properties of the service; the Profile on the other hand need not include all the functional properties. The process model is a specification of the ways a client may interact with a service. In the process model, there are three subclasses of the root class Process. These are Atomic process, Composite process and Simple process. As the name suggests, atomic processes are executed in one atomic step when invoked by the service requesters and return a message to the requester. Composite processes are compounds of simpler processes and can be decomposed through the control structures

such as Sequence, If-then-else, Split, Split-Join, and Repeat-Until. These control structures define the paths that the service requester can perform by sending and receiving messages. In contrast to the other subclasses, Simple processes can not be invoked, only constituting abstractions which can be realized either by an atomic process or a composite process. A simple process can be used as a perspective to other processes, and can provide a way to perform specific tasks, such as service composition.

3. The service supports a ServiceGrounding, which has a subclass called Grounding. The Grounding provides an interface to plug in WSDL descriptions and describes how to use a service. Grounding indicates how each atomic service can be invoked using a WSDL operation.

2.3 WEB SERVICE COMPOSITION

Today, with the increase in the number and usage of web services, finding and executing correct web services fulfilling the user needs become more difficult and critical. If there is a single web service which can satisfy the request alone, then locating that service is referred as web service discovery problem. However, when it is not possible to satisfy the functionalities requested by the user with a single web service, selecting and combining available web services in a correct order of execution in order to achieve that requested functionalities of user is referred as web service composition problem. There is a considerable amount of research in this web service composition problem [68].

Manual, semi-automated and automated solutions are proposed to web services composition problem. In manual solution composition takes place in the design process. User choses the web services, generates the workflow and then execute the generated manual composition. This solution may work fine as long as the web services in the created composition workflow do not change. Changes in web services may be a change of information provided by the service, unavailability of the web service, change in the location of the web service or change in the function names in the interface of the web service. In any one of these cases, it is unavoidable to change manual composition definition. In semi-automated solutions, again user constructs a workflow for the composition but in execution time, user is asked to select one of the alternative actual web services or asked to give constraints to filter the found services. In automated solutions, selection, composition and execution of web services are done in or-

der to satisfy user goals automatically. In this thesis, the focus is on automated web service composition.

2.3.1 AUTOMATED WEB SERVICE COMPOSITION

Automated web service composition is highly desirable since manual composition is a very complex and a challenging task. One reason for this is that the number of services available over the web has been increasing dramatically resulting in huge web service repositories to be searched [68]. Another reason is that web services are prone to frequent on the fly updates, thus the composition system needs to be informed about the update at runtime and the decision should be made based on the up to date information. Moreover, there does not exist a universal language to define and evaluate the web services since there are different models used to describe the services. Therefore, building composite web services with an automated tool is a very critical issue. There are two main approaches in automated web service composition which are namely, workflow based methods and AI planning based methods.

2.3.1.1 WORKFLOW BASED COMPOSITION TECHNIQUES

In web service composition, the work done is actually defining the control and data flow between web services which is very similar to workflow specifying the flow of work items. From this point, workflow management is proposed as a solution to web service composition problem. Static workflow generation and dynamic workflow generation are two methods used in workflow based web service composition.

Static workflow generation requires the abstract process model, most commonly as a graph, to be provided by the user before the start of planning. The abstract process model is composed of a set of tasks and their data dependencies. During the planning process, the query clause of each task in the abstract process model is used to find actual atomic web services in order to achieve the task. In this technique, the only part automatically performed is the selection and binding of atomic web services.

On the other hand, in dynamic composition technique, in addition to selection of atomic services, the process model creation is performed automatically too. The user only needs to specify constraints or preferences of the composition. More information on this technique can

be found in [23, 59].

2.3.1.2 AI BASED COMPOSITION TECHNIQUES

Planning problem is usually represented as finding the inner states between an initial state and a goal state. Given a set of goals and a set of process specifications, it is possible to derive a sequence of process instances which can accomplish those goals using AI planning methods. AI planning methods are widely used for the web service composition problem. The elements of solutions are represented as a set of available activities. Web Services can now be seen as activities and the planning can be used to create a composite service (plan) that satisfies the goals of a service requestor. Therefore, AI planning and web services composition problems are very similar, since both seek a (possibly partially) ordered set of operations that would lead to the goal starting from an initial state (or situation).

In order to apply AI planning methods to automated web service composition problem, services are represented as actions having parameters, preconditions, results and effects; and service composition is treated like a planning problem. With this approach each web service is first translated to a planning operator and the objective is expressed as a logical condition. Then the planner generates a plan which is essentially a sequence of web service instances; that is, a sequential composition that causes the goal condition to be true upon execution [78].

The AI planning methods are generally used when the user has no process model but has a set of constraints and preferences; hence the process model can be generated automatically by the program [68]. Using AI planning techniques for web services composition introduces some challenges [47] one of which is related to closed world assumption. The traditional planning systems assume that the planner begins with complete information about the world. However, in web service composition problem, most of the information (if it is available) must be acquired from the web services, or may require prior use of such information-providing services. In many cases, however, it is not feasible or practical to execute all the information-providing services up front to form a complete initial state of the world. Other challenges can be found in [78].

Considering the composition problem as an AI planning problem, different planners are proposed for the solution. A good survey about planning algorithms and their applications to

web service composition problem can be found in [65, 68].

Estimated-regression is a planning technique in which the situation space is searched with the guide of a heuristic that makes use of backward chaining in a relaxed problem space [54]. In this approach, the composition problem is seen as a PDDL planning problem. To apply this method to composition domain, an estimated-regression planner translates the composition problem to a PDDL planning problem and tries to solve it. A translator has been written which translates DAML-S and PDDL into each other [24]. In estimated regression planning approach Web Services are considered as actions of the planning domain. During the plan generation, a regression graph is constructed for each state starting from the initial situation, on which minimum cost heuristic is applied and the most feasible action whose preconditions are satisfied is selected.

Hierarchical Task Network (HTN) planning has been applied to the composition problem to find a collection of atomic processes that achieve the task [80, 79]. In these works, SHOP2 [60] is used as a domain-independent Hierarchical Task Network planning system that creates plans by iteratively decomposing the bigger tasks into smaller subtasks, until primitive tasks are found that can be performed directly. This approach is based on the relationship between OWL-S used for describing web services and Hierarchical Task Networks as in HTN Planning. OWL-S processes are translated into tasks to be achieved by the SHOP2 planner, and SHOP2 generates a sequence of the atomic services that achieves the desired functionality.

Graph search is another AI planning approach which relies on building a graph representation of all services available. The Graphplan [12] is the first general-purpose planner using graph search algorithms. Given a problem statement, Graphplan generates the graph using two kinds of levels, namely state levels and service levels. Service levels consist of the possible actions that have preconditions satisfied from the previous state level. State levels consist of the possible effects from the actions in previous service level. The graph is extended by state levels and service levels until all goal states are satisfied. After the graph is generated, Graphplan uses a backward search to extract a plan and allows for partial ordering among actions.

In [93], a graph based approach is proposed to achieve web service composition using inputs and outputs. In this work the nodes of the graph represent the available services and the edges of the graph encode whether one of the outputs of a service may serve another service

as one of its inputs. Edges are weighted according to a function of the execution time and the semantic similarity value of the associated input and output. The Bellman-Ford shortest-path dynamic programming algorithm is used to find the shortest path from the inputs of the user to the expected outputs which represent the best composition. The same idea has been investigated in an earlier work [50]. In this work, services are only allowed to have a single input and a single output to make the graph search simple. These approaches are similar to our approach such that they also generate the composition plan using inputs and outputs specified by the user. However, the major problem of these approaches is that they are not scalable with the number of available services.

The idea of using the event calculus in the context of web services has been investigated in some researches [70, 82, 91, 16, 29]. In [70] the event calculus has been used in verifying composed web services. Web services are coordinated by a composition process expressed in WSBPEL. In this work, an event-based approach is proposed for checking consistency of a business process, for mining the business process events, and for analyzing the process execution. WSBPEL constructs are translated into their corresponding Event calculus axioms in order to check its consistency for at both design time (static analysis) and runtime (dynamic analysis). In static verification, the WSBPEL process model is translated into Event Calculus predicates in order to check its consistency. This offers the ability to discover the potential flaws of such a process such as deadlocks, or unused branches in the control flow. Some interactions between web services may be dynamically specified at runtime, causing unpredictable interactions. In dynamic verification, verifying deviations with respect to the observed behavior of the process should be done in run-time. To provide this verification, the events that occur dynamically have been logged during the process execution, translated into event calculus predicates and verified. This approach is different from our approach, because here the aim is to verify a composed service, not generating the composition itself.

The work in [82] attempts to establish a link between agent societies and semantic web-services. In this work, issues of competence checking for agents operating in a global artificial society whose purpose is to organize complex services have been investigated. A controller agent performs a test which is formulated in order to decide a candidate agent should join a society according to a provided abstract description of their communicative competence. The event calculus which avoids abduction and stick to normal logic programs has been used for the competence checking.

In [91] an approach for formally representing and reasoning about commitments in the event calculus is developed. This approach is applied and evaluated in the context of protocols, which represent the interactions allowed among communicating agents. Traditionally, protocols have been specified using formalisms such as finite state machines, or Petri Nets, that only capture the legal orderings of actions. An approach for protocol specification that embodies the commitments of agents to one another has been developed in this work, since the semantic content of the actions is not captured, and the agents cannot reason about their actions. It is demonstrated that, agents that follow the specified protocols can decide on the actions they want to take by generating protocol runs with a planner. Event calculus is used for reasoning about actions and commitments. The changes of the world through the actions in a protocol, commitments, operations on them, and reasoning rules about them are represented using event calculus. In this approach, event calculus planner has been used to determine flexible execution paths that respect the given protocol specifications as a finite state machine, while in our approach event calculus is used to generate the web services composition automatically without giving an abstract composition plan or a workflow model.

In [16], the event calculus is used to model the coordination of web services in an already given specific web service composition by formally describing the interactions between the web services. In this work an event-based architecture is proposed for specifying and reasoning about composite events, which facilitates the detection of several inconsistencies that may arise when the coordinated web services are executed at run-time. On the other hand, in our approach, we use the abductive event calculus to generate the composition itself automatically.

An approach that uses the event calculus, like in our thesis, was proposed in [29]. In this work, automated web service composition was achieved by supplying a generic web service process definition. The generic processes are defined in OWL-S similar to a workflow definition. The generic process model is a compositions of atomic or composite services via control constructs such as Sequence, If-Then-Else, Split-Join, etc. According to these control constructs, the generic process is translated to the event calculus axioms. The inputs needed by the composition are asked to the user. Then, with the inputs supplied by the user, the translated event calculus axioms are given to the abductive theorem prover in order to generate the composition plans. The generated plans are presented to the user and the user executes by selecting one of the generated plans. This work is very close to our work with respect to

the used technologies and system the architecture. However, in our approach, web service composition is achieved by using only user specified inputs and outputs without any generic composition definition.

2.4 EVENT CALCULUS

The event calculus is a first-order logic framework that can be used to specify properties of dynamic systems which change over time. Such properties are specified in terms of events and fluents. An event in event calculus is something that occurs at a specific point in time and may change the state of a system. Fluents are conditions regarding the state of a system and are initiated and terminated by events [45]. The formulation of the event calculus is defined in first order predicate logic like the situation calculus. In situation calculus [53], a changing world is represented by a discrete ordered sequence of "snapshots", each representing the overall state of the world at a given instant. Because of this structure of situation calculus, it is very difficult to represent continuous change in world and concurrent events. This is the major difference between situation calculus and event calculus.

In the event calculus, every theory is composed of axioms which can have the main predicates listed in Figure 2.2. In a specific problem domain, an event calculus theory is defined with event axioms which actually include the descriptions of the initial states of the fluents, the effects of actions and which fluents hold at what times.

Initiates and *terminates* axioms are used to describe that a fluent is initiated or terminated by an event. *Initially* axiom is used to indicate the initial state of a fluent. The predicate *clipped* defines if a fluent F was terminated during a given time interval. Similarly *declipped* defines if a fluent F was initiated during a given time interval. These axioms can be defined as follows:

$$clipped(T1, F, T4) \leftrightarrow$$

$$(\exists E, T2, T3)[happens(E, T2, T3) \wedge terminates(E, F, T2) \wedge T1 < T3 \wedge T2 < T4]$$

$$declipped(T1, F, T4) \leftrightarrow$$

$$(\exists E, T2, T3)[happens(F, T2, T3) \wedge initiates(E, F, T2) \wedge T1 < T3 \wedge T2 < T4]$$

Predicate	Meaning
initiates(E,F,T)	Fluent F holds after event E at time T
terminates(E,F,T)	Fluent F does not hold after event E at time T
initially(F) *initially(¬F)*	Fluent F holds from the time of the initial state Fluent F does not hold from the time of the initial state
happens(E,T_1,T_2)	Event (action) E starts at time T_1 and ends at time T_2 where $T_1 \leq T_2$. Two argument version of happens predicate can be used for instantaneous events where *happens(E,T) ≡ happens(E,T,T)*
holdsAt(F,T) *holdsAt(¬F,T)*	Fluent F holds at time T Fluent F does not hold at time T
clipped(T_1,F,T_2)	Fluent F is terminated between times T_1 and T_2
declipped(T_1,F,T_2)	Fluent F is initiated between times T_1 and T_2

Figure 2.2: Essential event calculus predicates.

The *holds_At* axiom is used to query whether a fluent holds at a specific time. It defines whether or not a fluent holds since the initial state as follows:

$$holds\_At(F,T) \leftarrow initially(F) \wedge \neg clipped(T0,F,T)$$

$$holds\_At(\neg F,T) \leftarrow initially(\neg F) \wedge \neg declipped(T0,F,T)$$

The *holds_At* axiom defines whether or not a fluent holds at a given time as follows:

$$holds\_At(F,T) \leftarrow$$

$$happens(E,T1,T2) \wedge initiates(E,F,T1) \wedge \neg clipped(T1,F,T) \wedge T2 < T$$

$$holds\_At(\neg F,T) \leftarrow$$

$$happens(E,T1,T2) \wedge terminates(E,F,T1) \wedge \neg declipped(T1,F,T) \wedge T2 < T$$

If the *initially* axioms describing the initial states of the fluents, *happens* axioms describing the times of events and the *initiates* and *terminates* axioms describing the effects of events are known, it is possible to query whether a fluent holds at a given time by *holds_At* predicate. For example, let us assume the following axioms are given:

```
happens(ei,t1,t1).          initiates(ei,f,t1).
happens(et,t2,t2).          terminates(et,f,t2).
```

In these axioms fluent f is initiated by event *ei* at time $t1$, and fluent f is terminated by event *et* at time $t2$. Let us also assume that $t1 < t2$ is also given. It can be deducted that fluent f holds at time t when $t1 < t \leq t2$. It can also be deducted that fluent $\neg f$ holds at time t when $t2 < t$. Since *initially* axiom is not given in this example, none of fluents f or $\neg f$ holds at time t when $t \leq t1$.

2.5 ABDUCTIVE EVENT CALCULUS

Event calculus can be used, in both abductive and deductive reasoning. In deduction, "what's true when" is required given the "what happens when" and "what actions do" [77]. In abduction, on the other hand, "what happens when" is required given the "what actions do" and "what's true when". Event calculus has been used mainly for deductive reasoning especially in database applications. The use of event calculus for planning using abduction was first proposed by Kave Eshghi [28]. Then it was Shanahan who encoded the event calculus axioms in meta-level and wrote a meta-interpreter planning system in Prolog language in [74]. Shanahan's planning system, which is described as an abductive theorem prover (ATP), is a second order logical prover and will be used in this thesis to perform automated web service composition [5, 74].

For generating plans, ATP takes a list of goal clauses and tries to solve the goal list one by one using abduction. During the resolution, abducible predicates, before and happens, are stored in a residue to keep the record of the narrative. The narrative is a sequence of time-stamped events, and the residue keeping a record of the narrative is the plan.

In this thesis, the predicate *abduct* is used to denote the theorem prover. It takes a list of goal clauses and tries to find out a residue that contains the narrative. For each specific object level axiom of the event calculus, a meta-level abduct solver rule is written.

For example an object level axiom, in which AH is the head of the axiom and AB1 to ABN is the body definition of the axiom, in the form:

$$AH \leftarrow AB1 \wedge AB2 \wedge ... \wedge ABN$$

is translated to the following predicate form for the ATP:

```
axiom(AH, {AB1, AB2, ..., ABN})
```

During the resolution process axiom bodies are resolved by the *abduct* which populates the abducibles inside the residue. The used version of abduct solver in this thesis can be found in Appendix A. A simplified version is as follows:

```
abduct([],RL,RL,NL).
abduct([A|GL],CurrRL,RL,NL) <- abducible(A),
        NewRL = [A|CurrRL], consistent(NL,NewRL), abduct(GL,NewRL,RL,NL).
abduct([A|GL],CurrRL,RL,NL) <- axiom(A,AL),
        append(AL,GL,NewGL), abduct(NewGL,CurrRL,RL,NL).
abduct([not(A)|GL],CurrRL,RL,NL) <- irresolvable(A,CurrRL),
        abduct(GL,CurrRL,RL,[A(|)NL]).
```

In this definition GL, RL, NL, A and AL represent the goal list, the residue list, the residue of negated literals, axiom head and axiom body, respectively. The predicate *abducible* checks if the axiom is abducible. If an axiom is abducible, it is added to the residue; otherwise its body is inserted into the goal list to be resolved with the other axioms.

Negation as failure (NAF) technique is used for proving negative literals. When literals added to the residue, previously proved negated goals may no longer be provable. This situation may occur when negative literals were proven due to the absence of contradicting evidence; however the newly added literals might now allow the proof of the positive of literals, invalidating the previous negative conclusions. So, previously proven negated literals should be rechecked each time the residue is modified.

Whether the negated literal is resolvable or not is checked using *irresolvable* predicate. The negative literal in the query might also include a predicate which is not abducible. In this case it needs to be resolved with the axioms not the residue. This situation is explained in [74]. The *consistent* predicate is used for checking that none of the negated literals is resolvable with the current narrative residue using the predicate *irresolvable* for each negated literal.

For example, let us assume that we have the following specific axioms for a simple problem.

```
axiom(happens(e,T,T), [ ]).
axiom(initiates(e,f,T), [ ]).
```

These axioms mean that there is an instantaneous event e and that event initiates the fluent f when it occurs. We may submit query $abduct(holds\_At(f,t),RL)$ to investigate under which conditions the fluent f holds at time t. The residue list RL is bound to $[happens(e,t1,t1),\ t1 < t]$ which is a possible scenario in which the fluent f holds at time t.

CHAPTER 3

AUTOMATED WEB SERVICE COMPOSITION WITH THE EVENT CALCULUS

In this chapter we give a description of how the event calculus can be used as a method for automated web service composition. In order to generate the composition plan and get the desired outputs from web service executions, we need to create event calculus axioms and run the Abductive Theorem Prover (ATP). In the following sections, we mention about the representation of semantic web services in event calculus and plan generation using ATP.

3.1 REPRESENTATION OF WEB SERVICES IN THE EVENT CALCULUS

3.1.1 TRANSLATION OF WEB SERVICE DESCRIPTIONS

Semantic web services described by OWL-S ontology have inputs, outputs, preconditions and effects. In order to use abductive theorem prover successfully, all available web services in the repository must be translated into events descriptions. Inputs and outputs of the web service become the parameters of the translated event. In this translation, the general attitude is using the input and output names as event parameters. This is because fully semantic web services are not widely used and most of the services today are described by WSDL specification. This usage prevents us from understanding that most of the inputs and outputs are actually have the same semantic meaning even though they have different syntactic names. For example, let us assume that there are two web services namely ZipCodeFinder and FindZipCode. ZipCodeFinder has inputs with names NameOfTheCity and NameOfTheStreet respectively, and an output ZipCode. The second web service FindZipCode have two inputs CityName and StreetName and an output ZipCode. In non-semantic web services it is not possible to realize

that these two services serves the same purpose and have actually the same type of inputs unless some extra language processing techniques are applied. Moreover, such an extra effort will not be guaranteed to result in successful matchings. At this point, types of inputs and outputs become very important to understand their semantic meaning.

Semantic web service descriptions using OWL-S, gives us the opportunity to know that those two zip code finder services do the same job and have the same type of inputs and outputs, by defining the types ontologically. So in our system, while generating events from web service descriptions, we use input and output types as event parameters if they are ontologically described. Let us assume that there is an ontology called GeographicalTerms and it has a class called Address, dataTypeProperties streetName, cityName, zipCode associated with Address class. In the above zip code finding example, according to our system design, the parameters of the generated events become streetName, cityName and zipCode instead of NameOfTheStreet, NameOfTheCity, ZipCode or CityName, StreetName, ZipCode. Thus our abductive theorem prover knows that these two services are semantically same and can be an alternative of each other.

Each web service call is treated as an event and it is represented with a *happens* predicate. The parameters of the event get their actual values by jpl method predicate which is a precondition of the event and a call to the real web service. Every time the ATP resolves that event while generating the plans, the precondition jpl method is called and the inputs and outputs are retrieved by the actual web service call. How ATP makes that actual web service call using jpl method will be described in Chapter 5.

In addition to event occurrence axioms, the profile of the web service is translated into a set of effect axioms. While event axioms describe the execution and preconditions of the web service, effect axioms describe the behavior of the web service. In other words, we describe which parameters are inputs, which parameters are outputs. For example, let us assume a simple web service which returns the zip code of a given city and street. The example web service description in OWL-S and its corresponding event calculus model is as follows:
OWL-S description:

```
<process:AtomicProcess rdf:ID="FindZipCodeProcess">
        <service:describes rdf:resource="#FindZipCodeService"/>
        <process:hasInput rdf:resource="#streetName"/>
```

```
        <process:hasInput rdf:resource="#cityName"/>
        <process:hasOutput rdf:resource="#zipCode"/>
</process:AtomicProcess>

<process:Input rdf:ID="streetName">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &ontology;#StreetName
        </process:parameterType>
        <rdfs:label>Name of the Street</rdfs:label>
</process:Input>
<process:Input rdf:ID="cityName">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &ontology;#CityName
        </process:parameterType>
        <rdfs:label>Name of the City</rdfs:label>
</process:Input>
<process:Output rdf:ID="zipCode">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &ontolgy;#ZipCode
        </process:parameterType>
        <rdfs:label>Zip Code</rdfs:label>
</process:Output>
```

Event Calculus representation:

```
axiom(happens(pFindZipCode(StreetName, CityName, ZipCode), T1, TN),
[
        jpl_pFindZipCode(StreetName, CityName, ZipCode)
]).
axiom(initiates(pFindZipCode(StreetName, CityName, ZipCode),
known(zipCode, ZipCode), T),
[
        holds_at(known(streetName, StreetName), T),
        holds_at(known(cityName, CityName), T)
]).
```

In the above effect axiom, it is described that, if we know the inputs, which are streetName and cityName in this case, the findZipCode service makes the parameter ZipCode known. In the translation, there will be a separate effect axiom for every output of the web service. or instance if the web service has three outputs there will be three effect axioms for each output parameter. In order to resolve literals which are non-axiomatic assertions such as conditions or external calls abduct is extended to contain the following rule [6]:

```
abduct([G|GL], CurrRL, RL, NL) <- not( axiom(G,_)),  G,  abduct(GL, CurrRL, RL, NL)
```

In this rule G, GL, RL and NL denote, respectively, the non-axiomatic literal, the goal list, the narrative residue and the negation residue. By doing this, we say that the abductive theorem prover directly tries to prove the literal, when a non-axiomatic literal is encountered and if it is successful it continues with the rest of the goal list.

3.1.2 TRANSLATION OF INPUTS

In the proposed system, the user selects input types from a defined ontology list in order to get the desired outputs. The ontology list is populated from the web service descriptions in the repository. The ontologies which are used to declare input and output types of web services are retrieved while parsing the OWL-S descriptions to generate the event calculus axioms. The details of this process will be mentioned in Chapter 5. After selecting the inputs, the user supplies the values of those inputs. The selected inputs are ontologically defined, so that the user enters the values of these inputs according to their ontology definitions. The user can select an ontology class, a data value property or an object property from the listed ontology elements. If the user selects a data value property as an input, then the values of that property can be one of boolean, float, int, string, date, dateTime or time types. There is an “any” type, which is handled in our framework as string type. If the selected input is an object property, the user must select an instance of that object’s range class. For example user selects an object property of “Book” class, called “author”, which has a range class “Author”, then he or she must select one of the “Author” instances. If there is no “Author” instance in the defined ontology, user can create an instance of that type, by entering the necessary fields of that class. Another input type can be an ontology class, and in this case, the user again has to select an instance of that class or creates an instance then selects that instance. The

supplied input values are translated into event calculus axioms and these axioms describe the initial state of the world. The *initially* predicate is used in these event calculus axioms.

For instance, if the user selects cityName and streetName ontology data type properties, which have string type range values, as the input types and enters the values of these inputs like "Ankara" as cityName and "Koru" as streetName, the event calculus framework generates the following axioms:

```
axiom( initially(known(cityName, Ankara)), [] ).
axiom( initially(known(streetName, Koru)), [] ).
```

3.1.3 TRANSLATION OF OUTPUTS

Abductive theorem prover generates composition plans according to a goal state. Just like inputs, the user selects the output types from the ontology list, which s/he wants to get results of after the executions of web services. The selected outputs are translated into event calculus axioms and they are given as a query to ATP. The query consists of conjunctions of *holds_at* statements including the desired output parameters. For example, if the user wants to know two outputs namely outputA and outputB, the goal state created by the framework is as follows:

```
abdemo([holds_at(known(outputA,OutputA), t), holds_at(known(outputB,OutputB), t)], R).
```

ATP generates plans and returns the results of actual web service executions as OutputA and OutputB if the effect axioms are satisfied by the domain knowledge created by web service descriptions in the repository and the initial states supplied by the user as input values.

3.2 PLAN GENERATION WITH ATP

In our event calculus framework, after the web service descriptions are translated into corresponding axioms, the user specifies the values for inputs that s/he selected. In order to generate plans, the abductive theorem prover needs a goal state which is a conjunction of *known* axioms that are translated from the outputs selected by the user. Now our framework

returns the composition results according to the goal state. ATP resolves the event calculus axioms one by one and proves the goal state in order to generate the desired composition plan. To achieve this, there must be a parameter binding method that decides which services' outputs become which services' inputs. ATP does this binding using parameter names. Thus, the naming of axiom parameters becomes very important in order to generate correct composition plans. In every step during the resolution process, translated event axioms of web services with the actual web service calls as preconditions are executed. These web service calls are done by jpl methods for each web service. In other words, during the plan generation phase, actual web service executions are also done and the results of those executions are populated as parameters of the corresponding event axioms. ATP produces the plans as *happens* and *before* predicates. Multiple plans can be generated by the backtracking facility of Prolog. A plan is something like the following:

```
happens(service1([Inputs1], [Output1]), T1),
happens(service2([Inputs2], [Output2]), T2),
before(T1, T2), before(T2, T)
```

which means that in order to satisfy the goal state, service1 should be called first and then service2 should be called with their inputs. In totally ordered set of events, the service flow becomes sequential like in the above example. ATP can generate concurrent set of events, if the timestamps of events are equal.

```
happens(service1([Inputs1], [Output1]), T1),
happens(service2([Inputs2], [Output2]), T1), before(T1, T)
```

ATP can also generate partially ordered set of events, in which there is no relative ordering between the timestamps of events. In such a case, those events are assumed concurrent. In the following example, there is no ordering between T2 and T3, so service2 and service3 are considered as concurrent.

```
happens(service1([Inputs1], [Output1]), T1),
happens(service2([Inputs2], [Output2]), T2),
happens(service3([Inputs3], [Output3]), T3),
before(T1, T2), before(T1, T3), before(T2, T), before(T3, T)
```

If services return more than one result as their outputs, then ATP generates a different plan for each output result. Also while processing step by step, if there are more than one alternative solutions for one step, ATP generates different plans for each alternative service. For instance assume the user wants to find hotels close to his/her place and the price of one night accommodation for each hotel found. Assume that two hotels namely “Hilton” and “HolidayInn” are found by executing one web service FindHotels, and two web services return one night hotel accommodation prices for given hotels, respectively “FindHotelPrice1” and “FindHotelPrice2”. In this situation, ATP generates four different plans, which are as follows:

```
1.FindHotels --> "Hilton" and FindHotelPrice1 --> 100$
2.FindHotels --> "Hilton" and FindHotelPrice2 --> 105$
3.FindHotels --> "HolidayInn" and FindHotelPrice1 --> 130$
4.FindHotels --> "HolidayInn" and FindHotelPrice2 --> 135$
```

3.3 EXAMPLE

In this section, we will give an illustrative example to explain how the abductive event calculus framework can be used to solve a composition problem. The problem is learning the TL price of a book, giving only the name of the book. Let us assume there are four web services in our repository, which are namely BookFinder, BookPriceDollar, BookPriceTL and PriceConverterFromDollarToTL. BookFinder service gets the name of a book as input and returns the desired Book instance. BookPriceDollar service gets a Book instance as input and returns the dollar price of that book, similarly BookPriceTL service also gets a Book instance and returns the TL price of that book. PriceConverterFromDollarToTL service gets the dollar amount and converts it to its TL amount. The input and output parts of the OWL-S descriptions of these web services are as follows:

```
<process:AtomicProcess rdf:ID="BookFinderProcess">
        <service:describes rdf:resource="#BookFinderService"/>
        <process:hasInput rdf:resource="#BookName"/>
        <process:hasOutput rdf:resource="#BookInfo"/>
</process:AtomicProcess>
<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="BookInfo">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &book;#Book
        </process:parameterType>
        <rdfs:label>Book Info</rdfs:label>
</process:Output>
<process:AtomicProcess rdf:ID="BookPriceDollarProcess">
        <service:describes rdf:resource="#BookPriceDollarService"/>
        <process:hasInput rdf:resource="#BookInfo"/>
        <process:hasOutput rdf:resource="#BookPriceDollar"/>
</process:AtomicProcess>
<process:Input rdf:ID="BookInfo">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &book;#Book
        </process:parameterType>
        <rdfs:label>Book Info</rdfs:label>
</process:Input>
<process:Output rdf:ID="BookPriceDollar">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &price;#DollarPrice
        </process:parameterType>
        <rdfs:label>Book Price Dollar</rdfs:label>
</process:Output>
<process:AtomicProcess rdf:ID="BookPriceTLProcess">
        <service:describes rdf:resource="#BookPriceTLService"/>
        <process:hasInput rdf:resource="#BookInfo"/>
        <process:hasOutput rdf:resource="#BookPriceTL"/>
</process:AtomicProcess>
<process:Input rdf:ID="BookInfo">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &book;#Book
        </process:parameterType>
        <rdfs:label>Book Info</rdfs:label>
```

```
</process:Input>
<process:Output rdf:ID="BookPriceTL">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &price;#TLPrice
        </process:parameterType>
        <rdfs:label>Book Price</rdfs:label>
</process:Output>
<process:AtomicProcess rdf:ID="PriceConverterFromDollarToTLProcess">
        <service:describes
                rdf:resource="#PriceConverterFromDollarToTLService"/>
        <process:hasInput rdf:resource="#DollarPrice"/>
        <process:hasOutput rdf:resource="#TLPrice"/>
</process:AtomicProcess>
<process:Input rdf:ID="DollarPrice">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &price;#DollarPrice
        </process:parameterType>
        <rdfs:label>Dollar Price</rdfs:label>
</process:Input>
<process:Output rdf:ID="TLPrice">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &price;#TLPrice
        </process:parameterType>
        <rdfs:label>TL Price</rdfs:label>
</process:Output>
```

The corresponding event calculus axioms for the above services in Prolog are:

```
axiom(happens(pBookPriceDollar(Book, DollarPrice), T1, TN),
[
        jpl_pBookPriceDollar(Book, DollarPrice)
]).
axiom(happens(pBookPriceTL(Book, TLPrice), T1, TN),
[
        jpl_pBookPriceTL(Book, TLPrice)
]).
```

```
axiom(happens(pBookFinder(BookName, Book), T1, TN),
[
        jpl_pBookFinder(BookName, Book)
]).

axiom(happens(pPriceConverterFromDollarToTL(DollarPrice, TLPrice), T1, TN),
[
        jpl_pPriceConverterFromDollarToTL(DollarPrice, TLPrice)
]).
```

In addition to these axioms, effect axioms of the web services are generated by our event calculus framework. The generated effect axioms of these web services are as follows:

```
axiom(initiates(pBookPriceDollar(Book, DollarPrice),
known(dollarPrice, DollarPrice), T),
[
        holds_at(known(book, Book), T)
]).
axiom(initiates(pBookPriceTL(Book, TLPrice),
known(tLPrice, TLPrice), T),
[
        holds_at(known(book, Book), T)
]).
axiom(initiates(pBookFinder(BookName, Book),
known(book, Book), T),
[
        holds_at(known(bookName, BookName), T)
]).
axiom(initiates(pPriceConverterFromDollarToTL(DollarPrice, TLPrice),
known(tLPrice, TLPrice), T),
[
        holds_at(known(dollarPrice, DollarPrice), T)
]).
```

In order to generate compositions, we need to provide the initial situation with the help of user specified input values. In this case, the only input is book name. Because the bookName is a string type property, the user enters the name of the book. The given input value is translated into event calculus axiom. Let us assume user enters “Hamlet” as book name, then the generated axiom is as follows:

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

The last thing that should be done is to create the goal state which will be the query to ATP. In our example, the user wants to know the dollar and the TL price of book Hamlet. The query including goal state in this case is created as:

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

After the query is generated, ATP runs and returns the results according to the given event calculus axioms. In this example, ATP returns the prices and plans in a form of timestamped events as follows:

```
DollarPrice = 25 Dollar
TLPrice = 45 TL

happens(pBookPriceTL('HamletInstance', '45 YTL'), t1, t2)
happens(pBookFinder(Hamlet, 'HamletInstance'), t3, t4)
happens(pBookPriceDollar('HamletInstance', '25 Dollar'), t5, t6)
before(t2, t)
before(t4, t1)
before(t6, t)
before(t4, t5)
```

and,

```
DollarPrice = 25 Dollar
TLPrice = 43 TL
```

```
happens(pPriceConverter('25 Dollar', '43 TL'), t11, t12)
happens(pBookPriceDollar('HamletInstance', '25 Dollar'), t13, t14)
happens(pBookFinder(Hamlet, ' HamletInstance'), t15, t16)
before(t12, t)
before(t14, t11)
before(t16, t13)
before(t14, t)
```

ATP returns two different plans for the given inputs and outputs in this case. In the first plan, the BookFinder service is executed with the input "Hamlet" and returns the Book instance of Hamlet, HamletInstance. Then this output becomes the inputs of two other services, BookPriceDollar and BookPriceTL and these two services are called concurrently due to the absence of relative timestamped ordering between them. These services return 25 Dollar and 45 TL respectively, thus the goal state is reached and results are represented to the user. On the other hand, in the second plan, again BookFinder service is executed first in order to find the Book instance from a given name which is "Hamlet" in this example. Then the output of BookFinder service becomes the input of only BookPriceDollar service. BookPriceDollar service returns the dollar price, 25 Dollar, and one of the user desired output types becomes satisfied. The TL price is calculated as 43 TL by executing PriceConverter service with 25 Dollar which is the output of BookPriceDollar service. The goal state is again satisfied and results are shown to the user. In order to make the generated plans more readable, a user interface is implemented, in which the flow of web service executions are shown as a graph. The graphical representation of generated plans will be described in Chapter 5.

In this chapter, using event calculus as a solution to automated web service composition problem is explained. In order to generate the composition plan, the initial state of the world, defined actions and their effects in the problem domain and the required goal state should be known. Our framework gets these definitions from OWL-S descriptions of available web services in the repository and the user defined inputs and outputs. Then our system translates these definitions to event calculus axioms as described in this chapter. The translated axioms are given to abductive planner in Prolog and composition plans are generated using backtracing. An example that explains the translation and plan generation phases is given.

CHAPTER 4

PRECONDITIONS AND OUTPUT CONSTRAINTS

In this chapter, we give a description of how preconditions and user defined output constraints are included in our system. In the semantic web, web services are described with OWL-S which is an ontology for web services. In this ontology, web services have input, output, precondition and effects (IOPEs) which define the process model of a web service. In other words, IOPEs describe how to interact with the web service in detail. In the following sections, first we describe the usage of preconditions in semantic web services and how our system translates them into event calculus axioms. Then, we describe how users can define their output constraints and present the translation of those constraint definitions to the event calculus framework.

4.1 PRECONDITIONS

In a web service description, it must be specified how the web service will interact with clients or other software agents. The information transformation of this interaction is described by Input and Output properties. Inputs specify the types of instances to be sent to the service, and Outputs specify the types of responses to be sent by the service. However it is not possible to describe under which circumstances the web service provides its service with only inputs and outputs.

Preconditions are conditions that must be true for the web service in order to be executed. In most of the cases, preconditions are used to check whether the given inputs satisfy some conditions which are necessary for the service execution. This type of usage is generally seen in information providing services. For example, the “CurrencyConverter” service might have

a precondition which checks whether the given currency type is valid. But in world altering services, preconditions can be used to control the information space whether the required state is satisfied. An example of this kind of world altering service precondition can be a validation of the user's login state. In most of the e-commerce services, before the payment process, it is controlled whether the user is logged in; if s/he is, service goes on with the payment, if not, user is asked to log in. In OWL-S definition, Inputs and Outputs are subclasses of Parameter and their definitions in a semantic web service description are as follows:

```
<process:Input rdf:ID="InputID">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &ontology;#InputClass
        </process:parameterType>
</process:Input>
<process:Output rdf:ID="OutputID">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &ontology;#OutputClass
        </process:parameterType>
</process:Output>
```

However, preconditions do not have any specific format or class. They are kind of Expression properties. In OWL-S, they are treated as literals [51]. There are some formats that can be embedded into OWL language to define preconditions, such as SWRL (Semantic Web Rule Language) [39], KIF (Knowledge Interchange Format) [44] or PDDL (Planning Domain Description Language) [49]. Among these languages, SWRL is preferably used because of its more understandable grammar and built-in conditional operations. In our framework, we handled only SWRL expressions, but the others can also be handled in a similar approach.

For example, let us assume a web service "CalculateNumberOfDays" which takes startDate and finishDate as inputs, and returns the number of days between those two dates. In this service, there must be a control which compares input dates and returns true if the finishDate is greater than or equal to startDate. The service description should be as follows:

```
<process:AtomicProcess rdf:ID="CalculateNumberOfDaysProcess">
        <service:describes rdf:resource="#CalculateNumberOfDaysService"/>
        <process:hasInput rdf:resource="#StartDate"/>
```

```
        <process:hasInput rdf:resource="#FinishDate"/>
        <process:hasOutput rdf:resource="#Days"/>
        <process:hasPrecondition rdf:resource="#compareDates"/>
</process:AtomicProcess>

<process:Input rdf:ID="StartDate">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &date;#StartDate
        </process:parameterType>
        <rdfs:label>Start Date</rdfs:label>
</process:Input>

<process:Input rdf:ID="FinishDate">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &date;#FinishDate
        </process:parameterType>
        <rdfs:label>Finish Date</rdfs:label>
</process:Input>
<process:Output rdf:ID="days">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &date;#NumberOfDays
        </process:parameterType>
        <rdfs:label>Number Of Days</rdfs:label>
</process:Output>
<expr:SWRL-Condition
        rdf:ID="compareDates">
        <rdfs:label>compareDates</rdfs:label>
        <rdfs:comment>compares two dates</rdfs:comment>
        <expr:expressionBody
         rdf:parseType="Literal">
         <swrl:AtomList>
          <rdf:first>
           <swrl:BuiltinAtom>
            <swrl:builtin rdf:resource="&swrlb;#greaterThanOrEqual" />
            <swrl:arguments>
             <rdf:List>
              <rdf:first rdf:resource="&date;#FinishDate" />
              <rdf:rest>
               <rdf:List>
                <rdf:first rdf:resource="&date;#StartDate" />
```

```
                <rdf:rest rdf:resource="&rdf;#nil" />
               </rdf:List>
              </rdf:rest>
             </rdf:List>
            </swrl:arguments>
           </swrl:BuiltinAtom>
          </rdf:first>
          <rdf:rest rdf:resource="&rdf;#nil" />
         </swrl:AtomList>
 </expr:expressionBody>
</expr:SWRL-Condition>
```

This SWRL rule states that, startDate can not be greater than finishDate. The SWRL preconditions are translated as jpl methods with its owner service. So if a web service has a precondition, its corresponding event axiom has two jpl methods as preconditions. The first method handles the precondition and if the condition holds, it returns true. The second jpl method handles the real web service call, if the result of the first method is true. The translated methods in Prolog are as follows:

```
axiom(happens(pCalculateNumberOfDays(StartDate, FinishDate, NumberOfDays), T1, TN),
[
        jpl_pCalculateNumberOfDaysCondition('StartDate', 'FinishDate'),
        jpl_pCalculateNumberOfDays(StartDate, FinishDate, NumberOfDays)
]).
```

The precondition method defined in Prolog, gets inputs from the parameters of *happens* axiom and the validation operator from SWRL expression. It then makes a jpl call to the associated Java method to carry out the operation with supplied inputs. We do this operation in Java, because in our system all IOPEs are described with ontologies, so they are complex objects and can not be handled in Prolog. These Java methods will be mentioned in Chapter 5. The precondition method created in Prolog is as follows:

```
jpl_pCalculateNumberOfDaysCondition(Icondition1, Icondition2) :-
        jpl_new('tr.com.edu.metu.wsc.atlas.main.WebServiceInvocation', [], WSIa),
        jpl_call(WSIa, controlConstraint,
```

```
        [Icondition1, 'greater than or equals to', Icondition2], OutputArraya),
    jpl_is_true(OutputArraya).
```

4.2 USER DEFINED OUTPUT CONSTRAINTS

Semantic web service descriptions using OWL-S, can have preconditions in order to control in which circumstances the service should be invoked. But sometimes users want to restrict the generated plans according to their needs. In this case, even if the preconditions are satisfied, return values may not be what users really want. In addition to this, returned plans and their output values might include lots of irrelevant results that users do not want to see. To illustrate this, suppose the user wants to learn about the hotels close by and their prices. On the other hand, s/he wants to see the hotels and their prices if the found hotel's one night accommodation price is less than 100$. By giving this constraint, hotels which have accommodation price more than 100$ will not be presented to the user.

After the user selects the outputs s/he wants, according to their data type, s/he can specify a constraint on them, like specifying the inputs, the user can select a class, data type property or object property types as outputs. In our system, there are pre-defined operations for each type of output.

If the selected output is of type object property, the user can select either "equals to" or "not equals to" as an operation from the list and an instance of the range class of the selected object property. For example, let us assume the user wants to learn the price of a book by giving the name of the book. The ontology class of book has an object property called "author" which has a range class Author and the user wants to get the price of the book which has the author specified by her. In this case, the user can learn the price of the book written by, let us say Ayse Kulin, by giving an "equals to" constraint on the "author" object property and selecting "Ayse Kulin" from the Author instances.

If the type of the selected output is of data type property, according to the range of that property, operations differ. Such as, if the range of the selected property is string then the supported operations are "equals to", "not equals to", "starts with", "ends with", "contains" and "not contains". If the range of the selected property is an int or float then the supported operations are "equals to", "not equals to", "less than", "greater than", "less than or equals

to" and "greater than or equals to". Finally, if it is one of the date, dateTime or time types then the operations are "equals to", "not equals to", "before" and "after". These cases can be illustrated with examples, such as movies which have "Alien" in their name by using "contains" operation, movies which are made before 1987 by using "before" operation or movies which last more than 3 hours by using "greater than" operation.

If the selected output is of class type, then for each property of that class there becomes a constraint row with the operations described above. For example, assume that there is a Hotel class which has an object property "address" and a data type property "name". Object property "address" has a range class Address and data type property "hasName" has a range of string. So the user can either give a constraint on "address" property with the operations described for object property types, or can give a constraint on "name" data type property with string range by using the operations for string data type properties, or on both.

As in the SWRL preconditions defined in the OWL-S description of web services, we handle these type of output constraints given by the user in a similar approach. We translate output constraints as postconditions to event axioms of web services. But in this case, there might be more than one web service which will include the created jpl method as a postcondition. As an example, if the user gives an output constraint on TL price of a book; let us say "less than 30 TL", there can be one or more services that has "TLPrice" as an output parameter which will be included in the generated composition plan. Because the executions of web services take place while ATP is processing, we have to check the result returned by the web service whether it satisfies the given constraint or not. So the difference of this type of condition is that, it is located after the web service execution jpl method,

```
axiom(happens(pBookPriceTL(Book, TLPrice), T1, TN),
[
        jpl_pBookPriceTL(Book, TLPrice),
        jpl_pBookPriceTLCondition('BookPriceTL.TLPrice.amount','30')
]).

axiom(happens(pPriceConverter(DollarPrice, TLPrice), T1, TN),
[
        jpl_pPriceConverter(DollarPrice, TLPrice),
        jpl_pPriceConverterCondition('PriceConverter.TLPrice.amount','30')
```

```
]).
```

When the user gives a TLPrice constraint, two *happens* axioms include the postcondition because in our web service repository there are two web services having TLPrice as output. The reason behind the location of postcondition after the actual web service call is that, in this case we control the outputs instead of the inputs. So we have to know the returned output values before the constraint control. The input parameters passed to the postcondition method is a bit different in this case. 'BookPriceTL.TLPrice.amount' and 'PriceConverter.TLPrice.amount' are passed with the user constraint '30'. Our system makes the web service calls in the Java part, retrieve the results and sends them back to Prolog. The implementation details are described in Chapter 5. The results of web service executions are also kept in the system in order to achieve the constraint control. When the constraint control method is invoked with the parameters 'PriceConverter.TLPrice.amount', 'less than' and '30', the system gets the amount attribute of the TLPrice instance returned by the PriceConverter web service and checks whether the amount is less than 30. If it is, it returns true and the ATP goes on with its resolution. If it is false, then the whole *happens* axiom returns false because of this postcondition, and this service does not take place in the composition plan. The preconditions and the postconditions have the same method declaration which is as follows:

```
jpl_pServiceNameCondition(Icondition1, Icondition2) :-
        jpl_new('tr.com.edu.metu.wsc.atlas.main.WebServiceInvocation', [], WSIa),
        jpl_call(WSIa, controlConstraint,
          [Icondition1, 'operation', Icondition2], OutputArraya),
        jpl_is_true(OutputArraya).
```

In the event calculus framework proposed in this thesis, the descriptions of the web services that are available in the repository are translated to the event calculus axioms before the abductive theorem prover starts to generate the composition plan. In addition to service descriptions, the inputs and the output constraints provided by the user are translated to the corresponding event calculus axioms. The translated event calculus axioms using service descriptions do not change unless the web services in the repository change. On the contrary, the event calculus axioms that are translated using inputs and outputs specified by the user change each time the user wants to find out a composition plan. Therefore, in the proposed event calculus

framework, the Prolog code including the service descriptions, can be generated once and updated each time the user supplies inputs and output constraints dynamically, instead of generating whole Prolog code in every run. This facility can improve the system performance by reducing the translation costs. This problem can be handled as a future work.

The OWL-S descriptions of web services can have preconditions defined as expressions. There are different expression languages that can be used in OWL-S descriptions such as SWRL, KIF, PDDL. In this chapter, we show how SWRL preconditions are handled in the event calculus framework. They are translated as preconditions of web service axioms in Prolog. If the given precondition is satisfied, then the web service execution takes place. In addition to preconditions of OWL-S descriptions, users can supply output constraints as well. The supplied constraints are also translated into the event calculus axioms as postconditions.

CHAPTER 5

IMPLEMENTATION

In this chapter, we give a description of the implementation details of the proposed event calculus framework. The proposed system is expected to generate composition plans using user specified inputs and outputs only. Our approach is to make the user select inputs that s/he will enter to the system and outputs that s/he wants to get as results by using our user-friendly composition tool. First, we present the used technologies and libraries. Second, we represent the details of our tool and the usage of technologies in it. Then we give a case study in order to show the usage of our system. Finally, we explain how two web service composition algorithms are integrated in our event calculus framework.

5.1 TECHNOLOGIES

The event calculus framework is designed as a web-based application and implemented in Java and Prolog languages. The client side is implemented in JSF technology. In JSF codes, Ajax components are also used which help us partial rerendering of pages and doing some changes in client side without posting pages. In business tier, Java and Prolog languages are used together. All modules are implemented in a Web Project and deployed in a J2EE compatible application server, JBoss.

As open-source projects or libraries, MindSwap's OWL-S API is used for OWL-S and Ontology parsing, Java Universal Network/Graph Framework (JUNG) is used for the graphical visualization of the generated composition plans, JPL library is used for implementing the communication between Java and Prolog codes, JBoss' RichFaces component library is used for user-interface design.

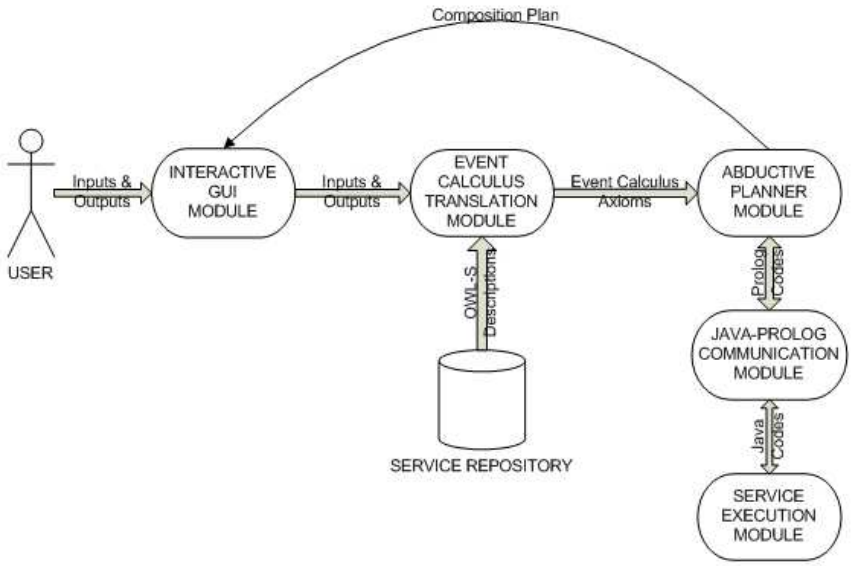

Figure 5.1: System Architecture.

5.2 SYSTEM ARCHITECTURE

In this section, the modules of our event calculus framework are presented. The user interacts with our system via interactive graphical user interface module. Users select the inputs and outputs, give constraints on selected outputs by using this GUI module. Moreover, the generated composition plans are presented to the user with this module. The event calculus translation module handles the translations of OWL-S descriptions to the event calculus axioms and the translations of user specified inputs and outputs to the event calculus axioms. The generated event calculus axioms are supplied to the abductive theorem prover (ATP) module for planning. The JPL module serves as the interaction module between service execution module which is implemented in Java and the ATP module which is implemented in Prolog. Web services are executed in service execution module and results of these service executions are returned to the ATP module via JPL module. Figure 5.1 shows the our system architecture graphically.

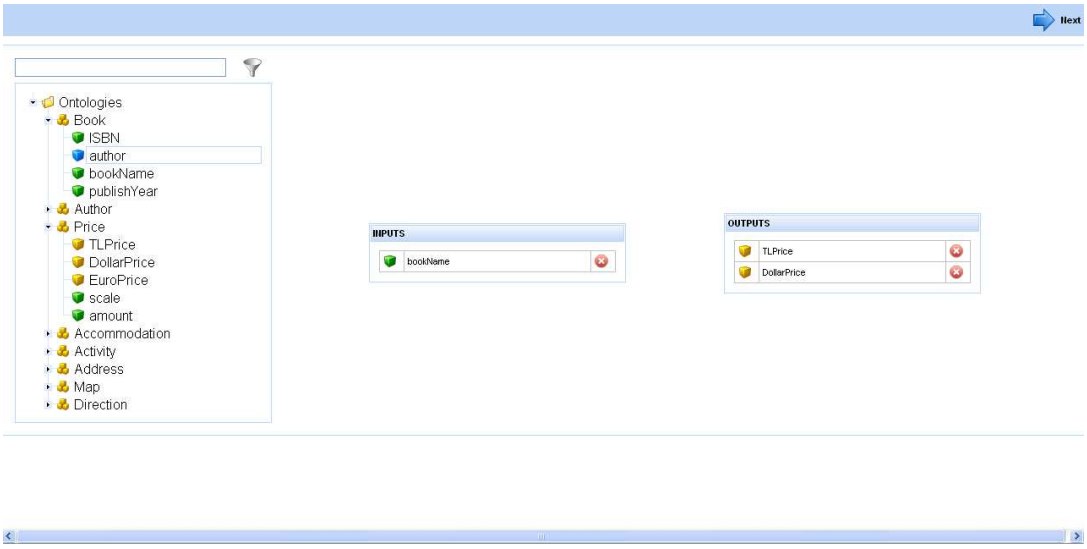

Figure 5.2: Input and output selection window.

5.2.1 INTERACTION WITH GUI

Since our application is a web-based application, users can access our application via their Internet browsers. When the user starts our application, using OWL-S API, all web service descriptions available in our repository are parsed and the ontologies that are used to declare input or output types in those service descriptions are retrieved. For example, if parsed OWL-S description of a web service has an input declaration as follows:

```
<process:Input rdf:ID="BookInfo">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &book;#Book
        </process:parameterType>
        <rdfs:label>Book Info</rdfs:label>
</process:Input>
```

then, our system adds the book ontology if it is not added before. After all service descriptions are parsed, found ontologies are parsed too using again OWL-S API, and they are presented to the user in a tree view as shown in Figure 5.2.

In the first page, the user can select the input and output types by drag&drop facility of RichFaces components. When the number of service descriptions in the repository increases,

the ontology resources will also increase. For this reason, there is a filtering mechanism on the ontology classes, which helps users to find ontology classes easily. The classes are filtered with the entered filter text.

The nodes in the ontology tree can be classes, object properties and data type properties. The nodes can have their subclasses or subproperties. The distinction between these types is done with image labels in front of the tree nodes. The yellow ones represent classes, the green ones represent data type properties and the blue ones represent object properties.

After selecting the inputs and the outputs, the user can continue with the next page in order to provide input values and output constraints by clicking the next button on the upper right corner of the page. Our application has a wizard style usage logic. Users can pass over pages by using next and previous buttons.

In the second page, it is expected from the user to supply the input values according to the selected input types and to give constraints on the selected output types if it is necessary. This page is split into three tabbed parts namely, “Input Values”, “Output Constraints” and “Create Instance” tabs. In the Input Values tab, the user can enter or select input values according to their types as shown in Figure 5.3. If the selected input is a class type input, then the user must select one of the instances of that class from a drop-down list. If the selected input is an object property type, again the user must select an instance of the object property range class from a drop-down list. If the selected input is a data type property, then the user enters the input value to that input’s value field. In this type of inputs, in order to become more user-friendly, different field types can appear in the page. When the selected data type property has int or float range values, the user enters in a number field which is shorter in length and the cursor is aligned to right. If selected data type property has string range value, then user enters the string into a standard text field. When selected data type property has date, dateTime or time range value, user can pick a date from a date field or can enter the value manually. When the selected data type property has a boolean range value, then user supplies the value by using a checkBox.

The second tab contains output constraint fields. There are selected outputs in a tree representation on the left side of the page. When the user clicks on an output node, the constraint fields appear on the right side of the page dynamically as shown in Figure 5.4. How users can define output constraints was described in Chapter 4.

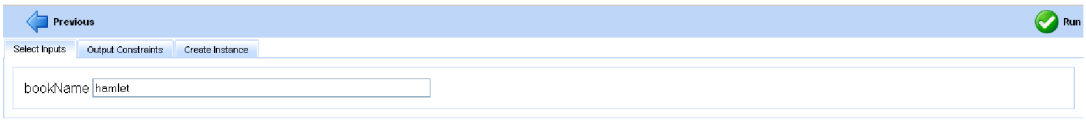

Figure 5.3: Input Values Tab.

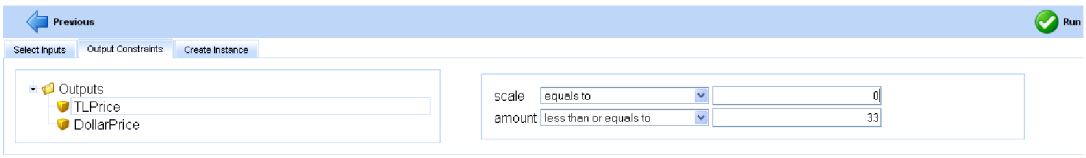

Figure 5.4: Output Constraints Tab.

The last tab includes components in order to create instances of ontology classes. In semantic web services, parameter types are ontologically defined and in semantic environments such as our framework, everything is actually done with class instances. These instances are used in communication with the abductive theorem prover. Users supply input values in order to define the initial state of the world. If the user selects a class type input, s/he must provide an instance of the selected class. In this event calculus framework, users can create instances by using this tab as shown in Figure 5.5.

After these steps, the user can run the ATP to generate the composition plan and return corresponding results. When the user presses the run button, the event calculus axioms in Prolog are generated and ATP starts to prove the goal state. In the resolution process of ATP, the *happens* axioms, which describe the web service execution steps, have jpl methods as preconditions in order to invoke the actual web services and return the results. In the following section, we describe how the communication between Prolog and Java is achieved.

5.2.2 COMMUNICATION VIA JPL

Java Interface to Prolog (JPL), is a library which handles the communication between Java and Prolog. Using JPL, we can execute Prolog codes from Java and get the results, and can invoke Java methods from Prolog and again get the results and use them in the next steps. In

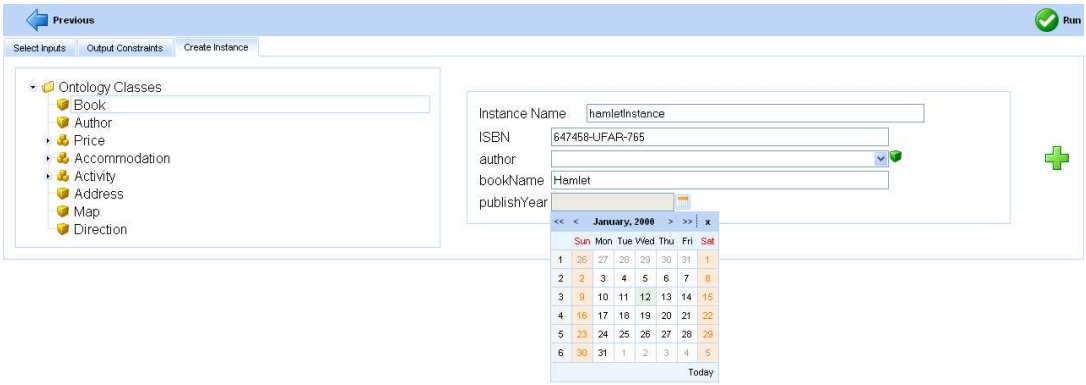

Figure 5.5: Create Instance Tab.

our event calculus framework, we use Prolog to Java calls in three cases which are invoking the web services and getting the results, controlling the preconditions defined with SWRL in the OWL-S description of web services and controlling the output constraints specified by user with the web service execution results. Web services in our repository, if it has no precondition or output constraint, is translated into their corresponding event axioms as follows:

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

The jpl method definition is as follows:

```
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),
        (OutputArray == @(null) ->  OutputList = [] ;
        jpl_array_to_list(OutputArray, OutputList)),
        [A] = OutputList,
        (A == @(null) ->  TempList1 = [] ;
        jpl_array_to_list(A, TempList1)),
```

```
        WholeList = [TempList1],
        member([Output], WholeList),
        true.
```

In jpl_pServiceName method, web service has an input and an output. These parameters are passed to the jpl method as arguments. The first row of the jpl method definition is for creating an instance of WebServiceInvocation Java class with constructor taking no arguments and the created object is assigned to a Prolog variable called WSI. These are done via using jpl_new method of JPL library. Then the input (which is only "Input" in our case), is populated into a Java array by using jpl_list_to_array method. Then the invokeService method of WebServiceInvocation java class is invoked with the serviceName and InputArray via jpl_call method. This invokeService method does the actual web service invocation and returns the results. The results are populated into OutputArray which is an argument of jpl_call method. Then this OutputArray is converted into a Prolog list by jpl_array_to_list method. Finally the converted output list is enumerated against the output of the service (which is only "Output" in this case), by using member operator. The whole jpl_pServiceName method returns true because it is a precondition of the above *happens* axiom and by returning true, it makes ATP to add this service into composition plan and to go on with resolution.

However, if the web service has a precondition defined in SWRL rule language in its description or has an output constraint given by the user, then in the *happens* axiom of the service, there becomes more than one precondition different from the above execution jpl method. Let us assume the user gives a constraint on the "Output" which is an int type output, such that it can not be greater than 10. Then the generated *happens* axiom becomes as follows:

```
axiom(happens(pServiceName(Input, Output), T1, TN),
[
        jpl_pServiceName(Input, Output),
        jpl_pServiceNameCondition(Output, '10')
]).
```

And the jpl method which handles the constraint validation is defined as follows:

```
jpl_pServiceNameCondition(Condition1, Condition2) :-
        jpl_new('tr.com.edu.metu.wsc.atlas.main.WebServiceInvocation', [], WSI),
```

```
        jpl_call(WSI, controlConstraint,
                [Condition1, 'less than or equals to', Condition2], ResultValue),
        jpl_is_true(ResultValue).
```

In this method definition, again an object instance of Java class “WebServiceInvocation” is created first via jpl_new method. Then the “controlConstraint” method of WebServiceInvocation class is invoked with three arguments and the returned result is assigned to ResultValue variable. The arguments are, the operation which is “less than or equals to” in this example, and the two values that will be controlled according to the given operation, Output and 10 in this case. After the controlConstraint method is executed and ResultValue is retrieved, this ResultValue is converted to Prolog boolean type by using jpl_is_true method and this boolean is returned from jpl_pServiceNameCondition. If it is true, ATP takes this service as a proved axiom, and goes on with other axioms.

This usage of jpl method describes from Prolog to Java communications and method calls. When ATP runs, these methods are necessary in order to do some jobs on Java side. But to start ATP and pass dynamically generated event calculus codes in Prolog, we also need interaction from Java to Prolog. JPL library also provides this facility. After all event axioms generated from service definitions and user inputs are completed, abductive theorem prover and these Prolog codes are consulted to a Prolog session. Now we can query this session from Java with the help of jpl. The query including the goal state, which is a conjunction of known predicates, is created using the outputs selected by the user and is fed to jpl. Then calling “allSolutions” method of the query object, JPL produces all possible composition plans for the desired outputs and these plans are put into a hash table for visualization processes. The Prolog calls from Java with the help of JPL are as follows:

```
Query consultATP =
  new Query("consult", new Term[] { new Atom(eventCalculusPlanner.getAbsolutePath()) });
consultATP.query();

Query consultDynamicAxioms =
  new Query("consult", new Term[] { new Atom(dynamicProlog.getAbsolutePath()) });
consultDynamicAxioms.query();

Query query = new Query(goalStateQuery);
results = query.allSolutions();
```

5.2.3 REPRESENTATION OF RESULTS

The results returned in the hash table are outputs of web services, *happens* axioms and *before* axioms. The *before* axioms declare the ordering of web services according to their invocation time. An example of a returned composition plan is as follows:

```
happens(service1([Inputs1], [Output1]), T1),
happens(service2([Inputs2], [Output2]), T2),
before(T1, T2),
before(T2, T)
```

This plan states that if service1 is first executed with inputs1, and then service2 is executed with inputs2, the desired outputs can be obtained. However this format probably would not be clear to end users. So, our framework shows the generated composition plans in a very user-friendly graphical representation. In this graphical representation, The Java Universal Network/Graph framework (JUNG) is used. JUNG is a very powerful graph representation library for Java. After we parse the generated plan, our system generates vertex objects for each web service in the plan with their timestamp values. After the vertices are created, using a time ordering algorithm as a post-processing, the edges between those vertices are also created. The mentioned algorithm starts with end vertices which are outputs in our case, and solves the time ordering between services in a backward processing. A sample plan representation is illustrated in Figure 5.6.

If the parameter passed between services is a class instance, the name of that instance appears on the edges in the generated graph. This is because, we pass method parameters from Java to Prolog with their names. In our framework, there can be very complex class definitions declared with their reference ontologies. In such a case, it is much more complex to represent those class instances in Prolog language than an object-oriented Java language. In our early example of finding a book, the web service BookFinder returns a book instance with a supplied book name. In that example, BookFinder returned a book object, but the name of the object instance was passed to Prolog. We bind those objects in our system, by keeping them in the resulting hash table. By doing this, we do not lose any data, and we can get the object instance anytime we need via retrieving it from the resulting hash table. Users can see only the instance names on the generated graph, but if they want to see all fields of the returned

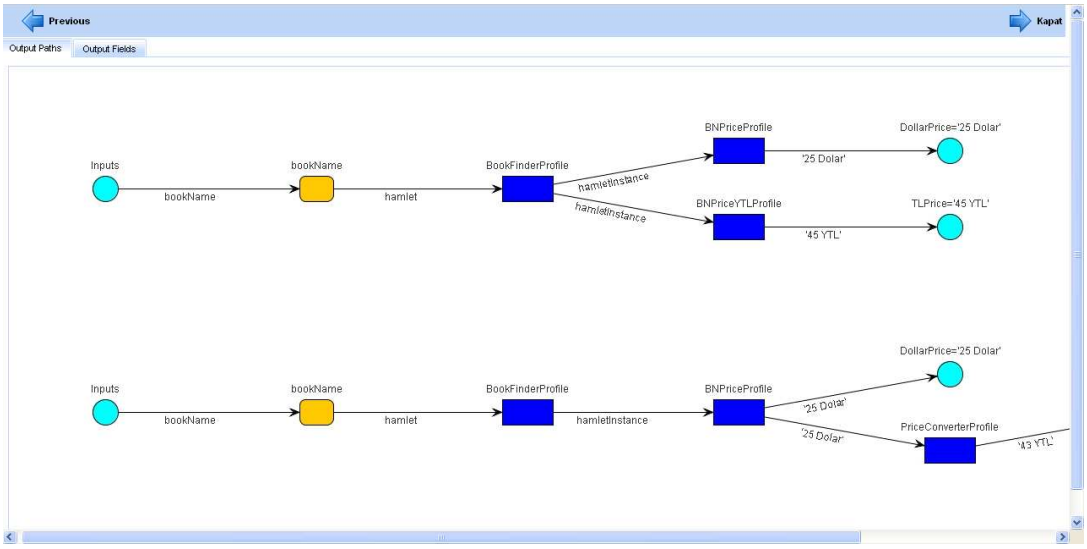

Figure 5.6: Graphical representation of generated composition plans.

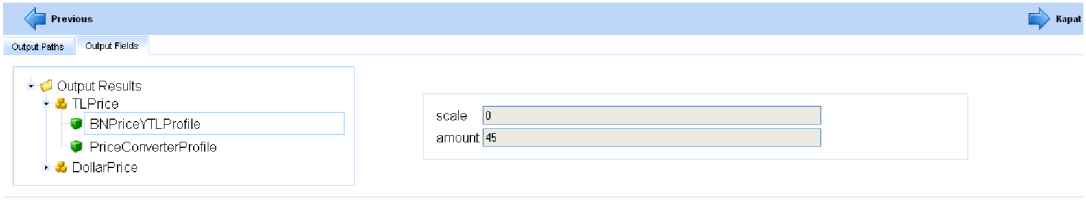

Figure 5.7: Detailed representation of output results.

instance, a different tab which is namely “Output Fields” is used. In this tab, there is a tree representation of outputs selected by the user. For each output node, the web services, which return that output value, are listed in the tree. To illustrate, in our example of finding a book, the user selected TLPrice as an output to find out the TL price of the found book. But there were two different services which returned TLPrice as their outputs. In our framework, when the user expands the TLPrice node in the output tree, s/he sees the two alternative services. Then by clicking on those web services, the user can see all fields of that instance as shown in Figure 5.7.

5.3 CASE STUDY: FINDING THE CLOSEST PREFERRED RESTAURANT

In this section, we aim to solve a composition problem given in [63] using our event calculus framework. The problem is about finding directions to the closest restaurant according to the user's food preference and also finding a map from the user's hotel address to found restaurant. Since there is no atomic web service that generates the desired outputs with the given inputs, there should be a composition of some services that satisfies the user needs. Assume that there are two available web services in our repository. One returns the address and name of the closest restaurant according to user supplied zip code and food preference. The other returns a map instance and direction, given start and finish addresses. IOPE parts of the web services' OWL-S descriptions are shown as:

```
<process:Input rdf:ID="ZipCode">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &travel;#zipCode
        </process:parameterType>
        <rdfs:label>Zip Code</rdfs:label>
</process:Input>

<process:Input rdf:ID="FoodPreference">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &travel;#foodPreference
        </process:parameterType>
        <rdfs:label>Food Preference</rdfs:label>
</process:Input>

<process:Output rdf:ID="RestaurantName">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &travel;#restaurantName
        </process:parameterType>
        <rdfs:label>Restaurant Name</rdfs:label>
</process:Output>

<process:Output rdf:ID="RestaurantAddress">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &travel;#Address
        </process:parameterType>
        <rdfs:label>Restaurant Address</rdfs:label>
```

```
</process:Output>
```

```
<process:Input rdf:ID="FromAddress">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &travel;#fromAddress
        </process:parameterType>
        <rdfs:label>From Address</rdfs:label>
</process:Input>

<process:Input rdf:ID="ToAddress">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &travel;#Address
        </process:parameterType>
        <rdfs:label>To Address</rdfs:label>
</process:Input>

<process:Output rdf:ID="Direction">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &travel;#Direction
        </process:parameterType>
        <rdfs:label>Direction</rdfs:label>
</process:Output>

<process:Output rdf:ID="Map">
        <process:parameterType rdf:datatype="&xsd;#anyURI">
                &travel;#Map
        </process:parameterType>
        <rdfs:label>Map</rdfs:label>
</process:Output>
```

In this case study, the ontologies used in service descriptions are retrieved as the initial preprocessing step. Then the user selects the inputs that s/he provides the values of and the outputs s/he wants to find out. So the user will select, foodPreference which is a data type property of Dinner class, zipCode which is a data type property of Address class and fromAddress which is an object property of Direction class as inputs. The input and output selections are shown in Figure 5.8. Map and Direction classes are the outputs of this example. All these classes and properties are defined in the “travel” ontology and can be found in Appendix B.

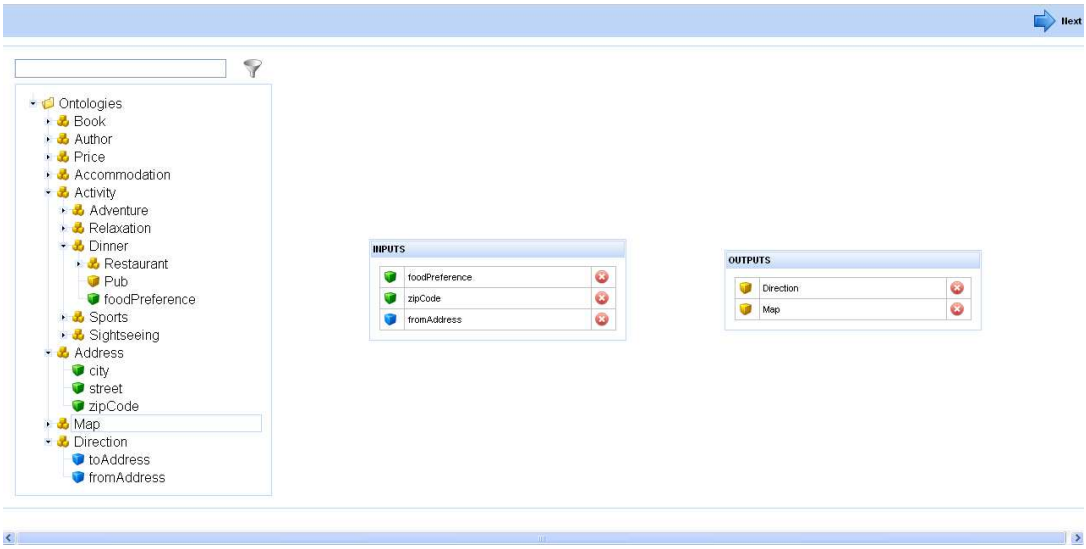

Figure 5.8: Input and output selection screen for the case study.

The *happens* axioms generated from those two web services are as follows:

```
axiom(happens(pFindDirection(Address, FromAddress, Map, Direction), T1, TN),
[
        jpl_pFindDirection(Address, FromAddress, Map, Direction)
]).

axiom(happens(pFindRestaurant(FoodPreference, ZipCode, Address, RestaurantName), T1, TN),
[
        jpl_pFindRestaurant(FoodPreference, ZipCode, Address, RestaurantName)
]).
```

The *initiates* axioms of those web services are created as follows:

```
axiom(initiates(pFindDirection(Address, FromAddress, Map,_),
known(map, Map), T),
[
        holds_at(known(address, Address), T),
        holds_at(known(fromAddress, FromAddress), T)
] ).

axiom(initiates(pFindDirection(Address, FromAddress, _,Direction),
known(direction, Direction), T),
```

```
[
        holds_at(known(address, Address), T),
        holds_at(known(fromAddress, FromAddress), T)
] ).

axiom(initiates(pFindRestaurant(FoodPreference, ZipCode, Address,_),
known(address, Address), T),
[
        holds_at(known(foodPreference, FoodPreference), T),
        holds_at(known(zipCode, ZipCode), T)
] ).

axiom(initiates(pFindRestaurant(FoodPreference, ZipCode, _, RestaurantName),
known(restaurantName, RestaurantName), T),
[
        holds_at(known(foodPreference, FoodPreference), T),
        holds_at(known(zipCode, ZipCode), T)
] ).
```

After selecting inputs and outputs, now the user can give values to the input parameters. Assume that, user is in Istanbul and stays at Swiss Hotel and wants to go to the closest Italian restaurant. But s/he does not know where it is and how to go to that restaurant from Swiss Hotel. Therefore, the user also wants to get the map and direction between Swiss Hotel and the closest Italian restaurant. According to these expectations, the user fills the input fields as "italian" for foodPreference, "34544" for zipcode. FromAddress is an object type property so the user should create an Address instance which is Swiss Hotel's address in this case as shown in Figure 5.10 and should select this created instance as fromAddress as shown in Figure 5.9.

The *initially* axioms which define the initial state of world are created with inputs provided by the user, as follows:

```
axiom( initially(known(fromAddress, addressofswisshotel)), [] ).
axiom( initially(known(foodPreference, italian)), [] ).
axiom( initially(known(zipCode, 34544)), [] ).
```

Figure 5.9: Giving the input values for the case study.

Figure 5.10: Creating an instance of “Address” class.

When the user clicks the run button, abductive theorem prover and dynamically created Prolog codes including the above event calculus axioms are consulted via JPL. The query is generated from the user specified outputs. Then the query is executed on the Prolog session to generate the composition plan. The query for finding the closest preferred restaurant problem is as follows:

```
abdemo([holds_at(known(map,Map), t), holds_at(known(direction,Direction), t)], R).
```

With the given inputs and outputs, ATP results in one composition plan which has the following *happens* and *before* axioms.

```
happens(pFindDirection('addressofmezzaluna', addressofswisshotel,
  'mapofmezzaluna', 'directionfromswisshoteltomezzaluna'), t1, t2)
happens(pFindRestaurant(italian, 34544, 'Mezzaluna', 'addressofmezzaluna'), t3, t4)
before(t2, t)
before(t4, t1)
```

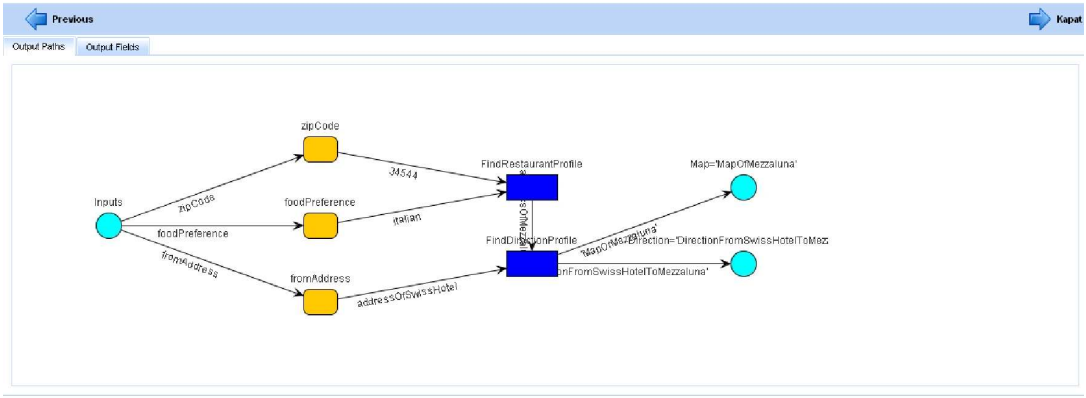

Figure 5.11: Generated composition plan and its JUNG representation.

This plan shows us if we first call FindRestaurant with given inputs and then call FindDirection with given Swiss Hotel address input and generated Mezzaluna address output, we can get the desired map and direction results. This plan is represented with JUNG as in Figure 5.11.

In this graphical representation we can see the input output parameters as labels on the edges. But as we mentioned before, if the output is a class instance then only the name of the instance appears on the edges. We can see the detailed output results in the “Output Fields” tab. In this case study, the map has two float type properties, namely latitude and longitude. If the selected output is a Map type output, the user can see the visual map in a modal panel appears on the screen. In this map representation, we used Google Map API to show it in a detailed manner as shown in Figure 5.12.

5.4 INTEGRATION WITH WORKFLOW FRAMEWORK

The event calculus has been used for automated web service composition problem before [29]. In that work, generating composition plans using generic web service composition templates described in OWL-S has been investigated. In this thesis, generating composition plans using only the user specified inputs and outputs is investigated. Both frameworks use abductive event calculus in order to solve the automated web service composition problem. In this thesis, we integrated both approaches under one application by adding one level abstraction on these two composition tools. The user can select which one s/he wants to use as a composition

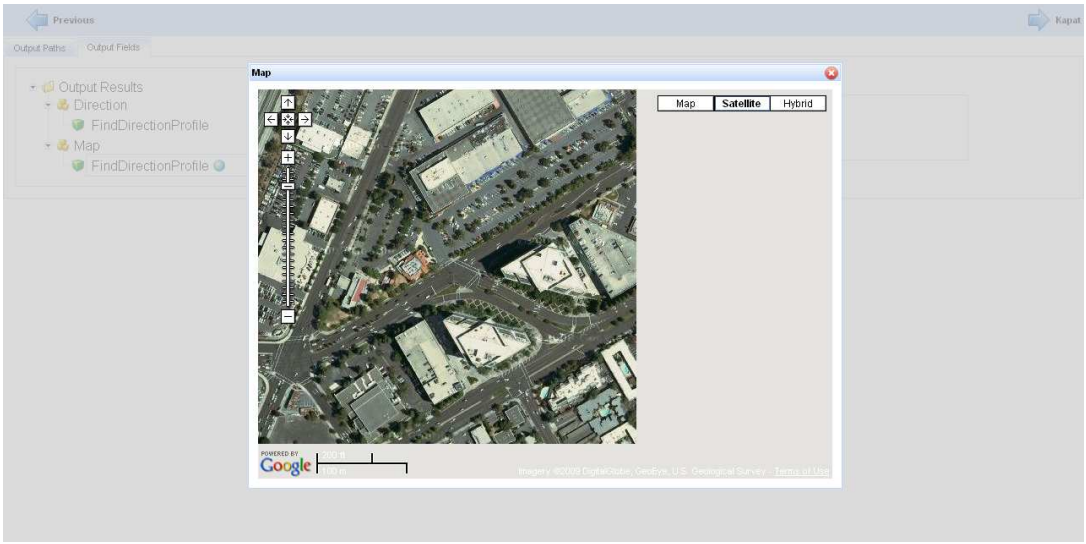

Figure 5.12: Google Map API used for map representation.

Figure 5.13: The method selection screen.

approach (see Figure 5.13). If the user has a generic composition definition in a local file and wants to explore actual web services and their return results according to supplied inputs, s/he selects the choice “Web Service Composition by Selecting Generic Composition Description File” and clicks on next button. If the user wants to make web service composition by giving a generic composition definition URL and wants to explore actual web services and their return results according to supplied inputs, s/he selects the choice “Web Service Composition by Giving Generic Composition Description URL” and clicks on next button. Finally, if the user wants get composition plans and execution results using only inputs and outputs without supplying any generic composition file or URL, s/he selects the choice “Web Service Composition by Selecting Inputs & Outputs” and clicks on next button. According to the selected composition choice, system goes on with our tool or the tool defined in [29].

The system architecture and the implementation details of our event calculus framework are presented in this chapter. The input and output selection, supplying the input values and giving

constraints on the outputs are explained in detail. The web service descriptions and the inputs and outputs supplied by the user are translated to the event calculus axioms. Then, abductive theorem prover resolves the given query and returns the composition plan. The generated plans are presented to the user with a graph which shows the necessary web services to fulfill the user requirements and their input and output parameters using vertices and directed edges. These steps are explained with the help of graphical user interfaces and the event calculus axioms. Furthermore, an illustrating case study is given in this chapter which is a composition problem given in [63]. Finally, the integration of our system with the tool proposed in [29] is presented.

CHAPTER 6

CONCLUSION

In this thesis, we have presented the usage of the event calculus as a method for the solution of automated web service composition problem. Given only the inputs and outputs specified by the user, our system can generate all possible composition plans with respect to the available services in the repository. Users can also give constraints on outputs in order to narrow down the created composition plans.

Web services are described using OWL-S in our system. The process model of OWL-S descriptions have the necessary information to achieve the proposed automated web service composition. The input, output, precondition and effect parameters of services described in the process model are used in order to represent events and their effects in the problem domain. The profile model of OWL-S descriptions are commonly used for service discovery. In our framework we assume all relevant services have been discovered and their service descriptions are in our local repository. The grounding part of OWL-S is used for service executions which are simulated in this thesis.

In this thesis it is shown that, when a goal state is given, the event calculus can find proper plans as web service compositions with the use of the abductive theorem prover. The event calculus is used as a logical formalism to describe the actions (web services) and their effects. The OWL-S descriptions of available web services in the repository are translated into corresponding event axioms in Prolog language. In this translation phase, instead of using input and output parameter names, parameter types are used with the help of semantic definition of OWL-S. In a fully semantic environment as assumed in this thesis, web services have parameter types as ontology resources such as properties or classes defined by the corresponding ontology files. In this approach, we can describe services via semantic meanings of input/out-

put parameters instead of their syntactic declarations. These ontology classes and properties used in web service descriptions are presented to the user in the input/output selection phase. The specified input values to the selected input types are then translated into event axioms that define the initial state of the world. Likewise, selected output types by user are converted to a goal state as conjunctions of event calculus predicates and given to the abductive planner. Then, the planner generates composition plans to reach the given goal state. In this plan generation phase, the abductive planner communicates with the execution module to call the actual web services and use the returned results in the remaining resolution steps. This execution part is simulated, due to the lack of those fully semantic web services published in the Internet. The generated composition plans are presented to the user as a graphical representation for ease of understanding and the details of results corresponding to user selected outputs are presented as form type fields.

In the planning and execution phases, the preconditions defined in the OWL-S service descriptions are also considered. These preconditions are added to converted event axioms in Prolog as preconditions of events on input parameters. If the preconditions of a service are satisfied, then the service is executed and takes place in the plan. In addition to service defined preconditions, users can give output constraints according to their needs in our system. After selecting the output types, users can define constraints on outputs and also on the attributes of outputs. While giving constraints, output types are taken into consideration and according to those types, users can select constraint operations such as “less than” for int type output, “before” for date type output or “starts with” for string type output. These constraints are added to event calculus axioms as postconditions on output parameters.

We also propose a complete event calculus framework for automated web service compositions, by integrating our tool with a previous work done in [29]. In that research, abductive capabilities of event calculus are also used to describe composition problem as a planning problem. The difference is that, an OWL-S description which includes generic composition model is provided to system, and actual web services are located to generate the composition plan in that method. Users can enter the input values of found services and execution takes after the planning. The integrated framework provides alternative composition approaches to users and they select one of the given methods. If the user has a composition model and wants to see the generated plan according to that model, he or she selects that option and goes on with the tool proposed in [29]. If the user does not have a generic composition plan and wants

to get a composition plan with respect to user specified inputs and outputs, he or she selects that option and uses our tool.

In this thesis, we showed that it is possible to represent web services which are described in OWL-S, in the event calculus domain. In the event calculus, properties of dynamic systems which change over time can be expressed easily. Because of this facility of the event calculus, concurrency and temporal ordering between events can be modeled easily. With the help of abductive planning capability, the event calculus is a suitable solution for web service composition problem.

As a future work, the simulated web service execution module can be replaced with the actual execution module. Also in our method, the execution of web services take place during the planning process. This is not a problem for information-retrieving services such as the ones we used in our case study, because execution phase is either during or after the planning process does not matter. However, in world-altering services, there might be cases that the user does not want to happen during composition steps. Thus, separating the planning and execution parts from each other and letting the user select and execute generated compositions after showing the alternative plans, is another future work.

The OWL-S descriptions of web services in the repository are translated into corresponding event axioms in Prolog language. The performance of the system can be tested when there are plenty of OWL-S descriptions in the repository.

When presenting the generated plans, the cost of the plans according to some benchmarks such as, the number of services in the plan or the execution time of the plan, can be considered and plans can be listed in an increasing order of cost. So, the user can see the plans with low costs on the top. When the number of plans are too much, how they can be optimized and presented to the user is another future work related with the presentation of plans.

REFERENCES

1. Andrews, T., Curbera, F., Dholakia, H., and Goland, Y., *Business Process Execution Language for Web Services, Version* http://www.ibm.com/developerworks/library/specification/ws-bpel/, 2003.

2. Arkin A., Askary S., Fordin S., Jekeli W., Kawaguchi K., Orchard D., Pogliani S., Riemer K., Struble S., Takaci-Nagy P., Trickovic I., and Zimek S., *Web Service Choreography Interface (WSCI) 1.0.* Published on the World Wide Web by BEA Systems, Intalio, SAP, and Sun Microsystems, 2002.

3. Arkin, A., *Business Process Modeling Language, Version 1.0*, Business Management Initiative, http://www.bpmi.org/, January 2009.

4. Au, T.C., Kuter, U., and Nau, D., *Web Service Composition with Volatile Information,* International Semantic Web Conference, 2005.

5. Aydin, O., *Automated Web Services Composition with the Event Calculus*, M.S. Thesis, METU, 2005.

6. Aydin, O., Cicekli, N.K., Cicekli, I., *Automated Web Services Composition with Event Calculus*, Proceedings of the 8th International Workshop in \Engineering Societies in the Agents World" (ESAW07), 2007.

7. Bachlechner, D., Lausen, H., Siorpaes, K., Fensel, D., *Web Service Discovery-A Reality Check*, Third Annual European Semantic Web Conference ESWC'06, 2006.

8. Baresi, Luciano, and Elisabetta Di Nitto. Test and Analysis of Web Services. Berlin [Germany] ; New York: Springer, pp. 266-303, 2007.

9. Bechhofer, S., Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-

Schneider, P.F., and Stein, L.A., *OWL Web Ontology Language Reference, W3C Recommendation 10 February 2004*, W3C Technical Reports and Publications.

10. Benatallah, B., Sheng, Q.Z., Ngu, A.H.H., and Dumas, M., *Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services*, Proc. of the 18th Int. Conf. on Data Engineering (ICDE'02), 2002.

11. Berners-Lee, T., Hendler, J., and Lassila, O., *The Semantic Web*, Scientific American Magazine, 2001.

12. Blum, A., and Furst, M., *Fast Planning Through Planning Graph Analysis*, Proceedings of the 14th International Joint Conference on Artificial Intelligence - IJCAI95, pp. 1636–1642, 1995.

13. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., and Orchard, D., *Web Services Architecture, W3C Working Group Note 11 February 2004*, W3C Technical Reports and Publications, http://www.w3.org/TR/ws-arch/, January 2009.

14. Carnegie Mellon University OWL-S API, http://projects.semwebcentral.org/projects/owl-s-api/, January 2009.

15. Casati, F., Ilnicki, S., and Jin, L., *Adaptive and Dynamic Service Composition in eFlow*, Proceedings of 12th Int. Conference on Advanced Information Systems Engineering(CAiSE), 2000.

16. Chen L., Yang X., Applying AI Planning to Semantic Web Services for workflow Generation, Proc. of the 1st Intl. Conf. on Semantics, Knowledge and Grid (SKG 2005).

17. Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S., *Web Services Description Language (WSDL) 1.1, W3C Note 15 March 2001*, W3C Technical Reports and Publications.

18. Curbera F., Goland Y., Klein J., Leymann F., Roller D, Thatte S., and

Weerawarana S., *Business Process Execution Language for Web Service (BPEL4WS) 1.0.*, Published on the World WideWeb by BEA Corp., IBM Corp. and Microsoft Corp., August 2002.

19. D. Tidwell, "Web Services—The Web's Next Revolution," IBM tutorial, 29 Nov. 2000

20. DAML's Bravo Air Process Example for OWL-S 1.1, http://www.daml.org/services/owl-s/1.1/BravoAirProcess.owl, January 2009.

21. DAML's Bravo Air Profile Example for OWL-S 1.1, http://www.daml.org/services/owl-s/1.1/BravoAirProfile.owl, January 2009.

22. David Martin, Massimo Paolucci, Sheila McIlraith, Mark Burstein, Drew McDermott, Deborah McGuinness, Bijan Parsia, Terry Payne, Marta Sabou, Monika Solanki, Naveen Srinivasan, and Katia Sycara. Bringing Semantics to Web Services: The OWL-S Approach. In Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004), San Diego, California, USA, 2004. URL: http://www.daml.org/services/owl-s/OWL-S-SWSWPC2004-CameraReady.doc.

23. Davulcu, H., Kifer, M., Pokorny, L., Ramakrishnan, C.R., Ramakrishnan, I.V., and Dawson, S., *Modelling and Analysis of Interactions in Virtual Enterprises*, RIDE, pp. 12-18, 1998.

24. Dejing Dou, Drew McDermott and Peishen Qi. Ontology Translation on the Semantic Web ,Journal on Data Semantics II, Springer-Verlag Lecture Notes in Computer Science no. 3360, Springer-Verlag, pp. 35-57 , 2005.

25. Dieter Fensel and Christoph Bussler. The Web Service Modeling Framework WSMF. Electronic Commerce Research and Applications, 1:113–137(25), Summer 2002. URL: http://www.swsi.org/resources/wsmf-paper.pdf.

26. Dustdar, S., and Schreiner, W., *A Survey on Web Services Composition*, Int. J. Web Grid Serv. 1 (1), pp. 1–30, 2005.

27. eCl@ss, The International Standard for the Classification of Products and Services, http://www.eclass-online.com/, January 2009.

28. Eshghi, K., *Abductive Planning with Event Calculus*, Proceedings of the 5th International Conference and Symposium on Logic Programming, MIT Press, pp. 562--579, 1988.

29. Esra Kırcı, Automatic Composition of Semantic Web Services with the Abductive Event Calculus, Masters Thesis, METU, September 2008

30. Fikes, R. E. and Nilsson, N. J., *STRIPS: a new approach to the application of theorem proving to problem solving*, Artificial Intelligence, 2(3-4): pages 189-208, 1971.

31. Fujii, K., and Suda, T., *Component Service Model with Semantics (CoSMoS): A new Component Model for Dynamic Service Composition*, Proceedings of Applications and the Internet Workshops (SAINTW'04), pp. 348-355, 2004.

32. Gardner, T., *An Introduction to Web Services*, Ariadne Issue 29, http://www.ariadne.ac.uk/issue29/gardner, January 2009.

33. Garofalakis, J., Panagis, Y., Sakkopoulos, E., and Tsakalidis, A., *Web Service Discovery Mechanisms: Looking for a Needle in a Haystack*, International Workshop on Web Engineering, 2004.

34. Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., and Wilkins, D., *PDDL: The Panning Domain Definition Language*, AIPS-98 Planning Committee, 1998.

35. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.J., Nielsen, H.F., Karmarkar, A., and Lafon, Y., *SOAP Version 1.2 Part 1: MessagingFramework (Second Edition)*, *W3C Recommendation 27 April 2007*, W3C Technical Reports and Publications, http://www.w3.org/TR/soap12-part1/, January 2009.

36. Haas, H., and Brown, A., *Web Services Glossary*, W3C Working Group Note 11 February 2004, W3C Technical Reports and Publications, http://www.w3.org/TR/ws-gloss/, January 2009

37. Huang, Y., and Walker, D.W., *Extensions to Web Service Techniques for Integrating Jini into a Service-Oriented Architecture for the Grid*, ICCS 2003, LNCS2659, pp. 254-263, 2003.

38. Hull, R., Hill, M., and Berardi, D., *Semantic Web Services Usage Scenario: e-Service Composition in a Behavior based Framework*, http://www.daml.org/services/use-cases/language/, January 2009

39. 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, 2003. Available at http://www.daml.org/2003/11/swrl/.

40. JPL - Java Interface to Prolog, http://www.swiprolog.org/packages/jpl/java_api/index.html, January 2009.

41. JUNG - Java Universal Network/Graph Framework, http://jung.sourceforge.net/, January 2009.

42. Karagoz, F., *Application of Schema Matching Methods to Semantic Web Service Discovery*, M.S. Thesis, Dept. of Computer Engineering, METU, Ankara, 2006.

43. Kautz, H., and Selman, B., *Planning as satisfiability*, In Proceedings of the 10th European Conference on Artificial Intelligence, 359–363. Wiley, 1992.

44. KIF. Knowledge Interchange Format: Draft proposed American National Standard (dpans). Technical Report 2/98-004, ANS, 1998. Also at http://logic.stanford.edu/kif/dpans.html.

45. Kowalski, R. A., and Sergot, M.J., *A Logic-Based Calculus of Events*, New Generation Computing, Vol. 4(1), pp. 67-95, 1986.

46. Kuster, U., Stern, M., and Konig-Ries, B., *A Classification of Issues and Approaches in automatic Service Composition*, 1st Int. Workshop on Engineering Service Compositions (WESC05) at ICSOC, 2005.

47. Kuter, U., Sirin, E., Parsia, B., Nau, D., and Hendler, J., *Information Gathering During Planning for Web Service Composition*, Proc. of ICAPS-P4WGS 2004, 2004.

48. Leymann, F., Web Service Flow Language (WSFL 1.0), IBM Software Group, http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, 2001.

49. M. Ghallab et al. PDDL-The Planning Domain Definition Language V. 2. Technical Report, report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

50. Mao, Z.M., Brewer, E.A., Katz, R.H.: Fault-tolerant, scalable, wide-area internet service composition. Technical Report UCB//CSD-01-1129, University of California, Berkeley, USA (2001)

51. Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N., and Sycara, K., *OWL-S: Semantic Markup for Web Services, W3C Member Submission 22 November 2004*, Acknowledged Member Submissions to W3C, http://www.w3.org/Submission/OWL-S/, January 2009

52. Maryland Information and Network Dynamics Lab, Semantic Web Agents Project (MindSwap) OWL-S API, http://www.mindswap.org/2004/owl-s/api/, January 2009.

53. McCarthy, J., *Situations, Actions and Casual Laws*, Stanford Artificial Intelligence Project: Memo 2, 1963.

54. McDermott, D., *Estimated-Regression Planning for Interactions with Web Services*, Sixth International Conference on AI Planning and Scheduling, AAAI

Press, 2002.

55. McIlraith, S. A., and Son, T.C., *Adapting Golog for Composition of Semantic Web Services*, Proceedings of Eighth International Conference on Principles of Knowledge Representation and Reasoning, pp. 482-493, 2002.

56. Miller, R., and Shanahan, M., *Some Alternative Formulations of the Event Calculus*, Computational Logic: Logic Programming and Beyond, Springer-Verlag, pp. 452-490, 2002.

57. Mithun Sheshagiri, "Automatic Composition and Invocation of Semantic Web Services", MastersThesis, UMBC, August 2004

58. Mueller, Erik T., *Commonsense Reasoning*, pp. 42-43, 2006.

59. Mueller, R., Greiner, U., and Rahm, E., *Agentwork: A Workflow System Supporting Rule-Based Workflow Adaptation*, Journal of Data and Knowledge Engineering, 2004.

60. Nau, D., Au, T.C., Ilghami, O., Kuter, U., Murdock, W., Wu, D., and Yaman, F., *SHOP2: An HTN Planning System*, JAIR Volume 20**,** pp. 379–404, 2003.

61. North American Industry Classification System, NAICS, http://www.census.gov/epcd/www/naics.html, January 2009.

62. OASIS, Organization for the Advancement of Structured Information Standards, http://www.oasis-open.org/, January 2009.

63. Oh, S.C., Lee, D., and Kumara, S., *A Comparative Illustration of AI Planning-based Web Service Composition*, ACM SIGecom Exchanges, 5(5), pp. 1-10, 2006.

64. Peer, J., *A PDDL Based Tool for Automatic Web Service Composition*, PPSWR' 04: Proceedings of Second International Workshop on Principles and Practice of Semantic Web Reasoning, pp. 149–163, 2004.

65. Peer, J., *Web Service Composition as AI Planning - a Survey,* Technical report, Univ. of St. Gallen, March 2005.

66. Pistore, M., Bertoli, P., Barbon, F., Shaparau, D., and Traverso, P., *Planning and Monitoring Web Service Composition*, Proc. of the 14th Int. Conf. on Automated Planning and Scheduling (ICAPS 2004), 2004.

67. *Process Ontology for OWL-S 1.1*, http://www.daml.org/services/owls/1.1/Process.owl, January 2009.

68. Rao, J., and Su, X., *A Survey of Automated Web Service Composition Methods*, Proceedings of First International Workshop on Semantic Web Services and Web Process Composition, pp 43-54, 2004.

69. Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S., *A Scalable Content-Addressable Network*, Proceedings of ACM SIGCOMM`01 Conference, pp. 161–172, 2001.

70. Rouached M., Perrin O., Godart C., Towards formal verification of web service composition, 4th Intl. Conference on Business Process Management, BPM 2006.

71. Rouached, M., and Godart, C., *An Event Based Model for Web Service Coordination*, 2nd International Conference on Web Information Systems and Technologies - WEBIST 2006, 2006.

72. Rowstron, A., and Druschel, P., *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems*, Lecture Notes In Computer Science, 2001.

73. Seppo Törma, Jukka Villstedt, Ville Lehtinen, Ian Oliver, Vesa Luukkala. Semantic Web Services — A Survey. Helsinki University of Technology, Laboratory of Software Technology, Helsinki, Finland, 2008

74. Shanahan, M.P., *An Abductive Event Calculus Planner*, The Journal of Logic Programming, Vol. 44(1-3), pp. 207--240, 2000.

75. Shanahan, M.P., *Event Calculus Planning Revisited*, Proceedings 4th European Conference on Plannning (ECP 97), Springer-Verlag Lecture Notes in Artificial Intelligence no. 1348, pages 390-402, 1997.

76. Shanahan, M., *Representing Continuous Change in the Event Calculus*, Proceedings of ECAI'90 Conference, Stockholm, pp. 598-603, 1990.

77. Shanahan, M.P., *The Event Calculus Explained*, Artificial Intelligence Today, Springer-Verlag Lecture Notes in Artificial Intelligence no. 1600, Springer-Verlag, pp. 409--430, 1999.

78. Sirin E., *Combining Description Logic Reasoning with AI Planning for Composition of Web Services*, PhD Thesis, Faculty of the Graduate School of the University of Maryland, , 2006.

79. Sirin E., Hendler J., Parsia B. Semi-automatic Composition of Web Services using Semantic Descriptions. Web Services: Modeling, Architecture and Infrastructure workshop in conjunction with ICEIS2003, 2002.

80. Sirin E., Parsia, B., Wu, D., Hendler, J., and Nau, D., *HTN planning for web Service Composition Using SHOP2*, Journal of Web Semantics, pp. 377–396, 2004.

81. Srinivasan, N., Paolucci, M., and Sycara, K., *An Efficient Algorithm for OWL-S Based Semantic Search in UDDI*, Lecture Notes in Computer Science, 2005.

82. Stathis K., Lekeas G., Kloukinas C., Competence checking for the global e-service society using games, In Proceedings of Engineering Societies in the Agents World (ESAW06), G. O'Hare, M. O'Grady, O. Dikinelli, and A Ricci (Eds).

83. Stoica, I., Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H., *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*, Proceedings of ACM SIGCOMM'01 Conference, pp. 149–160, 2001.

84. Su, X., and Rao, J., *A Survey of Automated Web Service Composition Methods*, In Proceedings of First International Workshop on Semantic Web Services and Web Process Composition, SWSWPC 2004, pp. 43-54, 2004.

85. Thatte, S., *XLANG: Web Services for Business Process Design*, Microsoft Corporation, http://www.gotdotnet.com/team/xmlwsspecs/xlang-c/default.htm, 2001.

86. UDDI, Universal Description, Discovery and Integration, *The UDDI Technical White Paper, September 2000, http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf, January 2009.*

87. Verma, K., Sheth, A., Miller, J., and Aggarwal, R., *Semantic Web Services Usage Scenario: Dynamic QoS based Supply Chain*, http://www.daml.org/services/use-cases/architecture/, January 2009.

88. *Web Ontology Language*, http://en.wikipedia.org/wiki/Web_Ontology_Language, January 2009.

89. Wilk, J., Russo, A., and Cunningham, M.J., *Dynamic Workflow Pulling the Strings*, Distinguished Project (MEng), Department of Computing, Imperial Collage London, 2004.

90. WonderWeb OWL Ontology Validator, http://www.mygrid.org.uk/OWL/Validator/, January 2009.

91. Yolum P., Singh M., Reasoning About Commitments in the Event Calculus: An Approach for Specifying and Executing Protocols, Annals of Mathematics and AI, Vol:42(1-3), 2004.

92. Zhang, J.F., and Kowalczyk, R., Agent-based Dis-graph Planning Algorithm for Web Service Composition, International Conference on Computational Inteligence for Modelling Control and Automation and International Conference on Intelligent Agents Web Technologies and International Commerce (CIMCA'06), pp. 258, 2006.

93. Zhang, R., Arpinar, I.B., Aleman-Meza, B.: Automatic composition of semantic web services. In: Proc. of the 2003 Int. Conf. on Web Services (ICWS'03), Las Vegas, NV, USA. (2003)

APPENDIX A

ABDUCTIVE THEOREM PROVER

The Abductive Theorem Prover [5] used in this thesis is provided below.

```
abduct(GL, RL) <-  abduct(GL, [], RL, []).

abduct([], RL,RL,N).
abduct([holdsAt(F,T) | GL], CurrRL, RL, NL) <-
      not(F=neg(_)),  axiom(initially(F), AL),
     irresolvable(clipped(0,F,T), CurrRL, NL),
     append(AL, GL, NewGL),
     abduct(NewGL, CurrRL, RL, [clipped(0, F, T) | NL]).
abduct([holdsAt(neg(F), T) | GL], R1, R3, N1, N4) <-
     axiom(initially(neg(F)), AL),
     irresolvable(declipped(0, F, T), CurrRL, NL),
     append(AL, GL, NewGL),
     abduct(NewGL, CurrRL, RL, [declipped(0, F, T) | NL]).
abduct([holdsAt(F, T) | GL], CurrRL, RL, NL) <-
     not(F=neg(_)),   axiom(happens(E, T1, T2), GLHappens),
     axiom(initiates(E, F, T1), GLInitiates),
     consistent(happens(E, T1, T2), CurrRL, NL, R1),
     consistent(<(T2, T), R1, NL, NewRL),
     irresolvable(clipped(T1, F, T), NewRL, NL),
     append(GLInitiates, GL, GL1),  append(GLHappens, GL1, NewGL),
     abduct(NewGL, NewRL, RL, [clipped(T1, F, T) | NL]).
abduct([holdsAt(neg(F), T) | GL], CurrRL, RL, NL) <-
     axiom(happens(E, T1, T2), GLHappens),
     axiom(terminates(E, F, T1), GLTerminates),
     consistent(happens(E, T1, T2), CurrRL, NL, R1),
```

```
     consistent(<(T2, T), R1, NL, NewRL),
     irresolvable(declipped(T1, F, T), NewRL, NL),
     append(GLTerminates, GL, GL1),  append(GLHappens, GL1, NewGL),
     abduct(NewGL, NewRL, RL, [declipped(T1, F, T) | NL]).
abduct([G | GL], CurrRL, RL, NL) <-
     abducible(G),  axiom(G, AL),
     consistent(G, CurrRL, NL, NewRL),
     append(AL, GL, NewGL),
     abduct(NewGL, NewRL, RL, NL).
abduct([G | GL], CurrRL, RL, NL) <-
     not(abducible(G)),  axiom(G, AL),
     append(AL, GL, NewGL),
     abduct(NewGL, CurrRL, RL, NL).

abducible(<(_, _)).
abducible(happens(_, _, _)).
```

APPENDIX B

TRAVEL ONTOLOGY

```
<?xml version="1.0" ?>
- <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns="http://localhost:8080/ESODENEME_ATLAS_WEB/owl/travel.owl#"
 xmlns:owl="http://www.w3.org/2002/07/owl#"
 xmlns:dc="http://purl.org/dc/elements/1.1/"
 xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xml:base="http://localhost:8080/ESODENEME_ATLAS_WEB/owl/travel.owl">
- <owl:Ontology rdf:about="">
  <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
          An example ontology for tutorial purposes.
  </rdfs:comment>
  <owl:versionInfo rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
          1.0 by Holger Knublauch (holger@smi.stanford.edu)
  </owl:versionInfo>
  </owl:Ontology>
- <owl:Class rdf:ID="Accommodation">
  <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
          A place to stay for tourists.
  </rdfs:comment>
  </owl:Class>
  <owl:Class rdf:ID="Activity" />
  <owl:Class rdf:ID="Direction" />
- <owl:Class rdf:ID="BunjeeJumping">
- <rdfs:subClassOf>
  <owl:Class rdf:ID="Adventure" />
```

```
  </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Address" />
- <owl:Class rdf:ID="Sightseeing">
- <owl:disjointWith>
  <owl:Class rdf:ID="Sports" />
  </owl:disjointWith>
- <owl:disjointWith>
  <owl:Class rdf:ID="Relaxation" />
  </owl:disjointWith>
- <owl:disjointWith>
  <owl:Class rdf:about="#Adventure" />
  </owl:disjointWith>
  <rdfs:subClassOf rdf:resource="#Activity" />
  </owl:Class>
- <owl:Class rdf:about="#Sports">
  <rdfs:subClassOf rdf:resource="#Activity" />
- <owl:disjointWith>
  <owl:Class rdf:about="#Adventure" />
  </owl:disjointWith>
- <owl:disjointWith>
  <owl:Class rdf:about="#Relaxation" />
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Sightseeing" />
  </owl:Class>
- <owl:Class rdf:ID="Surfing">
  <rdfs:subClassOf rdf:resource="#Sports" />
  </owl:Class>
- <owl:Class rdf:ID="Hiking">
  <rdfs:subClassOf rdf:resource="#Sports" />
  </owl:Class>
- <owl:Class rdf:ID="Dinner">
  <rdfs:subClassOf rdf:resource="#Activity" />
  </owl:Class>
- <owl:Class rdf:ID="Pub">
  <rdfs:subClassOf rdf:resource="#Dinner" />
  </owl:Class>
```

```
  <owl:Class rdf:ID="Map" />
- <owl:Class rdf:ID="Museums">
  <rdfs:subClassOf rdf:resource="#Sightseeing" />
  </owl:Class>
- <owl:Class rdf:ID="Sunbathing">
- <rdfs:subClassOf>
  <owl:Class rdf:about="#Relaxation" />
  </rdfs:subClassOf>
  </owl:Class>
- <owl:Class rdf:about="#Relaxation">
  <rdfs:subClassOf rdf:resource="#Activity" />
  <owl:disjointWith rdf:resource="#Sports" />
  <owl:disjointWith rdf:resource="#Sightseeing" />
- <owl:disjointWith>
  <owl:Class rdf:about="#Adventure" />
  </owl:disjointWith>
  </owl:Class>
- <owl:Class rdf:ID="Safari">
- <rdfs:subClassOf>
  <owl:Class rdf:about="#Adventure" />
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="#Sightseeing" />
  </owl:Class>
- <owl:Class rdf:ID="Yoga">
  <rdfs:subClassOf rdf:resource="#Relaxation" />
  </owl:Class>
- <owl:Class rdf:ID="Hotel">
  <rdfs:subClassOf rdf:resource="#Accommodation" />
- <owl:disjointWith>
  <owl:Class rdf:ID="BedAndBreakfast" />
  </owl:disjointWith>
- <owl:disjointWith>
  <owl:Class rdf:ID="Campground" />
  </owl:disjointWith>
  </owl:Class>
- <owl:Class rdf:about="#BedAndBreakfast">
  <owl:disjointWith rdf:resource="#Hotel" />
```

```
- <owl:disjointWith>
  <owl:Class rdf:about="#Campground" />
  </owl:disjointWith>
  <rdfs:subClassOf rdf:resource="#Accommodation" />
  </owl:Class>
- <owl:Class rdf:ID="Restaurant">
  <rdfs:subClassOf rdf:resource="#Dinner" />
  </owl:Class>
- <owl:Class rdf:about="#Campground">
  <rdfs:subClassOf rdf:resource="#Accommodation" />
  <owl:disjointWith rdf:resource="#BedAndBreakfast" />
  <owl:disjointWith rdf:resource="#Hotel" />
  </owl:Class>
- <owl:Class rdf:about="#Adventure">
  <rdfs:subClassOf rdf:resource="#Activity" />
  <owl:disjointWith rdf:resource="#Sports" />
  <owl:disjointWith rdf:resource="#Sightseeing" />
  <owl:disjointWith rdf:resource="#Relaxation" />
  </owl:Class>
- <owl:ObjectProperty rdf:ID="toAddress">
  <rdfs:domain rdf:resource="#Direction" />
  <rdfs:range rdf:resource="#Address" />
  </owl:ObjectProperty>
- <owl:ObjectProperty rdf:ID="address">
  <rdfs:range rdf:resource="#Address" />
- <rdfs:domain>
- <owl:Class>
- <owl:unionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#Activity" />
  <owl:Class rdf:about="#Accommodation" />
  </owl:unionOf>
  </owl:Class>
  </rdfs:domain>
  </owl:ObjectProperty>
- <owl:ObjectProperty rdf:ID="fromAddress">
  <rdfs:domain rdf:resource="#Direction" />
  <rdfs:range rdf:resource="#Address" />
```

```
  </owl:ObjectProperty>
- <owl:DatatypeProperty rdf:ID="latitude">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#float" />
  <rdfs:domain rdf:resource="#Map" />
  </owl:DatatypeProperty>
- <owl:DatatypeProperty rdf:ID="longitude">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#float" />
  <rdfs:domain rdf:resource="#Map" />
  </owl:DatatypeProperty>
- <owl:DatatypeProperty rdf:ID="foodPreference">
  <rdfs:domain rdf:resource="#Dinner" />
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string" />
  </owl:DatatypeProperty>
- <owl:DatatypeProperty rdf:ID="street">
  <rdfs:domain rdf:resource="#Address" />
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string" />
  </owl:DatatypeProperty>
- <owl:DatatypeProperty rdf:ID="hotelName">
  <rdfs:domain rdf:resource="#Hotel" />
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string" />
  </owl:DatatypeProperty>
- <owl:DatatypeProperty rdf:ID="restaurantName">
  <rdfs:domain rdf:resource="#Restaurant" />
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string" />
  </owl:DatatypeProperty>
- <owl:FunctionalProperty rdf:ID="city">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string" />
  <rdfs:domain rdf:resource="#Address" />
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty" />
  </owl:FunctionalProperty>
- <owl:FunctionalProperty rdf:ID="zipCode">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string" />
  <rdfs:domain rdf:resource="#Address" />
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty" />
  </owl:FunctionalProperty>
  </rdf:RDF>
```