AUTOMATED WEB SERVICE COMPOSITION WITH EVENT CALCULUS

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

BY

ONUR AYDIN

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

SEPTEMBER 2005

Approval of the Graduate School of Natural Applied Sciences

Prof. Dr. Canan Özgen
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Ayşe Kiper
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Nihan Kesim Çiçekli
Supervisor

Examining Committee Members :

Prof. Dr. Varol Akman	(Bilkent Univ.):	
Assoc. Prof. Dr. Nihan Kesim Çiçekli	(METU, CENG):	
Assoc. Prof. Dr. İlyas Çiçekli	(Bilkent Univ.):	
Assoc. Prof. Dr. Ferda Nur Alpaslan	(METU, CENG):	
Dr. Ayşenur Birtürk	(METU, CENG):	

ABSTRACT

AUTOMATED WEB SERVICE COMPOSITON WITH EVENT CALCULUS

Onur Aydın
M.Sc., Department of Computer Engineering
Supervisor: Assoc. Prof. Dr. Nihan Kesim Çiçekli

September 2005, 123 pages

As the Web Services proliferate and complicate it is becoming an overwhelming job to manually prepare the Web Service Compositions which describe the communication and integration between Web Services. This thesis analyzes the usage of Event Calculus, which is one of the logical action-effect definition languages, for the automated preparation and execution of Web Service Compositions. In this context, planning capabilities of Event Calculus are utilized. Translations from Planning Domain Description Language and DARPA Agent Markup Language to Event Calculus are provided to show that Web Services can be composed with the help of Event Calculus. Also comparisons between Event Calculus and other planning languages used for the same purposes are presented.

Keywords: Event Calculus, Web Service Composition, Planning

ÖZ

OLAY CEBİRİ İLE OTOMATİK ÖRÜN SERVİSİ KOMPOZİSYONU

Onur Aydın
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doçent. Dr. Nihan Kesim Çiçekli

Eylül 2005, 123 sayfa

Örün Servisleri yaygınlaştıkça ve karmaşıklaştıkça Örün Servisleri'nin birbirileriyle olan iletişimleri ve entegrasyonunu tanımlayan Örün Servisi Komposizyonları'nın manuel olarak hazırlanması zorlaşmıştır. Bu çalışma, mantıksal eylem-etki tanımlama dillerinden biri olan Olay Cebiri'nin otomatik olarak Örün Servisi Kompozisyonu oluşturulması ve koşulması için kullanımını analiz etmektedir. Bu çerçevede komposizyonun oluşturulması esnasında Olay Cebiri'nin planlama becerileri kullanılmıştır. Örün Servisleri'nin Olay Cebiri yardımıyla kompozisyonun oluşturulabildiğini göstermek için Planlama Sahası Betimleme Dili ve DARPA Etken İşaretleme Dili'nden Olay Cebiri'ne çevrimler verilmiştir. Ayrıca Olay Cebiri ile benzer amaçlarla kullanılan diğer planlama dilleri arasında karşılaştırmalar sunulmuştur.

Anahtar Kelimeler: Olay Cebiri, Örün Servisi Kompozisyonu, Planlama

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Semantic Web Services

Semantic Web can be defined as the information which can be perceived and classified according to some reference system. One of its most basic forms is the meta-content tag in the Hypertext Markup Language, namely HTML, which identifies keywords about certain and crucial facts of the web site. This information is very beneficial for the web searching engines and they are utilized in great extend along with other types of inputs that are mined from web sites.

One of the drawbacks of this meta-information is the lack of any common language or standard. Since they could be chosen from any combination of words actually they do not contain reliable and exact information about the owner Web Site. Hence it is hard to reach a whole understanding even with a certain level language processing. For that reason they are not machine readable. The ultimate goal of the Semantic Web is to transform the Web into a medium through which data can be shared, understood, and processed by automated tools [37].

The same problem is inherited by the constructs that depend on Web communication technologies and one of them is Web Services. Web

Services can be described as a set of related functionalities that can be programmatically accessed through the Web protocols [37]. These protocols are derived from Extensible Markup Language (XML) [9] and they supply the information about how to call the service but similarly they do not define the semantics either. Web Service Description Language (WSDL) [42] is the one that has been currently used widespread and it is describing binding of the web service. On the other hand, extension efforts on this language is now being consolidated into consortiums Semantic Web Service Initiative (SWSI) and DARPA Agent Markup Language (DAML) Program Committee to make Web Services semantically understandable by the automated systems. Efforts of these groups yielded to a series of languages: DAML+OIL [5], DAML-S [6] and finally OWL-S [27].

These languages have a well-defined semantics and enable the markup and manipulation of complex taxonomic and logical relations between entities on the Web [25]. OWL-S, [27] which is mainly derived from WSDL and Resource Description Framework (RDF) [30] structures, supports ontological classification of the elements based on predefined taxonomies and definition of complex service interactions with flow control constructs (eg: loops, conditionals, splits). Most interesting and innovative side of OWL-S, which is one of the main motives of this thesis, is that it contains logical structures which are very common to many logical planning languages like Planning Domain Definition Language (PDDL) [36] and STRIPS [10].

Almost in parallel, companies in industry (Microsoft, IBM, BEA Systems) also created a series of standards for the communication of web services in complicated behavioral scenarios and encompassing publication of these efforts is Business Process Execution Language for Web Services (BPEL4WS) [3]

1.2 Problem Definition

The distribution of the functions of the business through web services helped a lot to integrate services of different companies. Business to business (B2B) or business to commerce (B2C) communication is eased because of a defined standard on data exchange through Web Services. This flexible architecture allows solid breakdown of the job to be done by different sites.

However as the web applications flourished and the number of Web Services increase another difficulty appeared in the horizon. Enterprise application integrators now should find the correct Web Service that meets the demands of the customer while building the applications or even when the customer enters requests during runtime on a finished application.

In such a dynamic domain, automatic integration or composition of web services would be helpful since the unknowns of the demands are too much or too diverse. This brings us to the problem of Automatic Web Service Composition.

Initially the name of the problem is thought to be encapsulating the whole problem however as it leaves the hypothetical grounds and finds some practical uses [24, 41, 1, 14] in the field, it is seen that it implies only a stage of the whole process of which all stages are listed below.

1.2.1 Web Service Discovery

It is the first stage of Automatic Web Service Composition. It involves machine understanding of the request interpreted in plain human language. It is the least studied part of the whole process.

One of the usage areas of Natural Language Processing on this problem can be to decompose the user query into keywords that represent the service function. Afterwards, as it is done in [26], an ontological search can be performed to match the functional elements with Web Services registered on databases like Universal Description Discovery & Integration (UDDI) [38] registries or Electronic Business XML (ebXML) [7] registries. Languages like OWL-S (or predecessors DAML-S) are used in accordance with UDDI to support semantic service searches [19].

1.2.2 Web Service Composition

Automatic composing or integrating the candidate Web Services is the main problem at hand. Composed Web Services could be compared to programming language functions where the atomic Web Service calls are the internal functions. It encapsulates a specific function hence it has an input and most of the time if executed it has a world-altering effect. If it is desired it might generate an output to be used by an upper composition in a nested composition.

Automatic construction of a composition involves automatic selection of discovered services and composition of the interoperable services to meet the desired functionality. The outcome is a ready to execute plan that generates a solution to the task at hand.

In semi automatic composition of Web Services the interoperation of services is checked by the system however the composition is defined by the user [33]. In generic compositions, a template composition for a specific problem at hand is developed and user constraints are added to the composition to obtain a user specific composition instance at runtime [24].

1.2.3 Composition Execution

The execution step is simply running the composition inside the domain that it has been defined. Web Services are called successively according to the composition flow and outcomes of the previous calls obeying the axiomatic rules of the system.

In this stage outputs and effects of called Web Services guide the flow of execution. For example if a flight traveling fails an alternative traveling with a rented car might be arranged by the composition if such an outcome is expected by the composition. Similarly user inputs might alter the flow. If there is a maximum price constraint for a traveling then train or bus traveling might be arranged instead of airplane.

1.3 Scope

In the scope of the thesis, automatic composition and execution using a logical formalism, Event Calculus [16] has been analyzed. The Web Service Discovery Problem is beyond of the scope of this thesis since it is not directly related with the Event Calculus.

Abductive planning of Event Calculus is used to show that when atomic services are available, composition of services that would yield the desired effect is possible. Secondly it is shown that Event Calculus might be used as a plan executer when a generic composition is available. An Abductive Planner implementation of Event Calculus [32] is extended to be used for both directions of the problem: for plan generation (abduction) and plan execution (deduction).

1.4 Organization of The Thesis

The rest of the thesis is organized as follows. Chapter 2 gives insight information about current technologies, the Web Services Composition problem and techniques used to solve the problem. In Chapter 3, Event Calculus as a logical formalism and its implementation are explained in. In Chapter 4, abduction of Event Calculus on solving the Web Service

Composition problem is presented. In Chapter 5, Event Calculus is shown to be usable for the execution of Web Service Compositions. Finally in Chapter 6, conclusions and possible future extensions are listed.

CHAPTER II

BACKGROUND INFORMATION

2.1 Web Services

Being the world's largest distributed network World Wide Web flourished in huge amounts and became an important part of every society. Being a vast connected network, some other application areas are arisen other than being just an information sharing system for end users.

One of the biggest application areas of the web is the Web Services. They are the programmatic interfaces which are made available over the web. Atomic Web Services can have inputs and outputs like programming functions. After they are executed the application which has been bound to the web interface generates the outputs and effects. For example a hotel reservation system registers the reservation to the hotel automation system (effect) and returns the successful completion confirmation of operation (output) when reservation information and credit card information (input) are supplied externally.

2.1.1 Types of Web Services

According to the task performed inside, Web Services can be grouped under two categories. First group is the Information Collecting/Gathering Web

Services and second group is the World-Altering Web Services. A Web Service that returns the raster image of the supplied field coordinates in raw data is an example of Information Collecting Web Service.

On the other hand, a hotel reservation service described above is a World-Altering one since the current state of the domain is altered after completion of a successful reservation. In practice, if a Web Service has an effect on its domain after the execution it is accepted to be a World-Altering Web Service.

2.1.2 Web Service Definition Language (WSDL)

Web Services are mostly defined and described in a text based language called Web Service Definition Language (WSDL) in a WSDL document. It is a widely accepted standard and it is fully based on XML. WSDL is regulated by a group of companies and the latest valid specification [42] is distributed also by W3C Consortium as a note without any legal endorsement.

2.1.2.1 WSDL Document Sections

A WSDL Document is composed of six sections as it is given in the Table 2-1.

Table 2-1 WSDL Document Sections

Section	Description
types	data type definitions used to describe the messages
message	abstract definition of parameters of operations
portType	abstract set of operations where an operation is composed of an input message and output message.
binding	concrete protocol and specification used for operations and messages for a specific portType
service	used to aggregate set of related ports

These sections are repeated more than once if there are more entities to be specified in the document. For example if there are two *service* sections then it means two Web Service definitions exist in the document. An abstract hierarchical relationship of the entities appearing in a WSDL document is given in Figure 2-1.

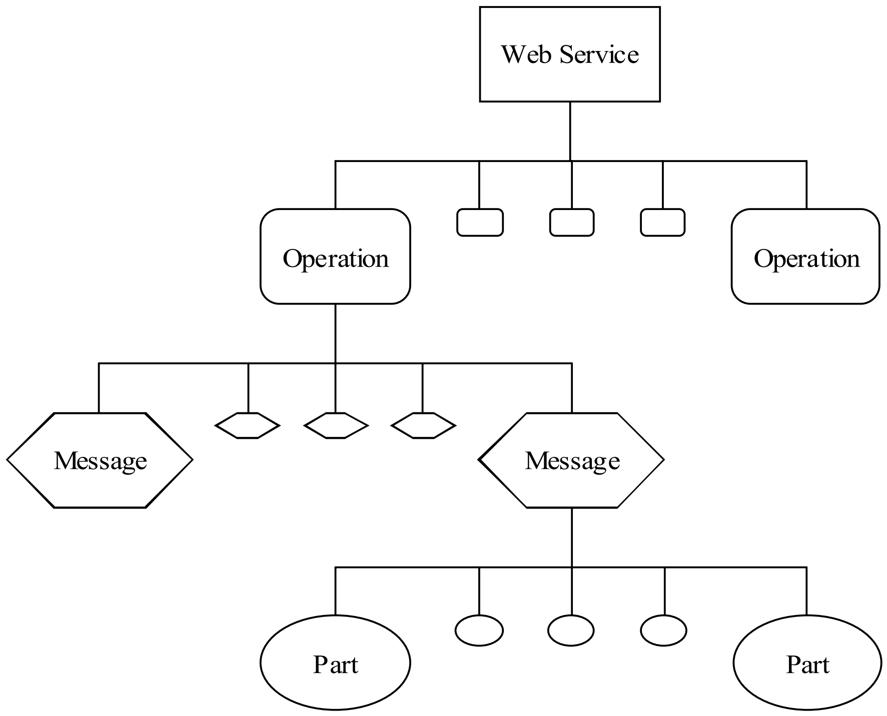

Figure 2-1 Hierarchical Representation of WSDL Entities

In the *types* section, data types that are referenced in *message* sections are given. For maximum interoperability and platform independence, WSDL recommends using the XML Schema Definition (XSD) documents (via imports) or syntax since it is globally accepted common way of defining complex types.

In the *message* section abstract definitions of *operation* parameters are listed. Messages can have multiple *part*s where each should be related with a *type*.

In the *portType* section *operation*s of the Web Service are given in an abstract list. Under each *operation* its parameters are given and each of these parameters should be related to a *message*. Also their direction to the operation as input or output should be given. Based on the direction of parameters four common *operation* types are defined as in Table 2-2.

Table 2-2 Operation Types

Operation Type	Operation Contains
One-way	Only input messages
Request-response	First input and then output messages
Solicit-response	First output and then input messages
Notification	Only output messages

In the *binding* section format and protocol details for *operation*s and *message*s are given for a specific *portType*. In other words abstract operation set is bound to a concrete protocol within this section.

Finally in *service* section *port*s, which are the endpoint addresses of the *binding*s, are listed. Each *binding* should be connected to an at least one address through *ports*.

2.1.2.2 WSDL Protocol Bindings

WSDL allows incorporation of several data exchange formats as bindings which are also web based communication standards. It allows generic structure for service definition as well as ability to wrap already established applications in these formats. Web Services can be communicated through Hypertext Transfer Protocol (HTTP) Get/Post, Simple Object Access Protocol (SOAP) and Multipurpose Internet Mail Extensions (MIME).

An example WSDL document taken from [42] containing Request-response operation about stock quotation given in APPENDIX A. It contains a single operation *GetLastTradePrice* which takes stock symbol and returns its stock price.

2.2 PDDL

PDDL [36] is first appeared in the Artificial Intelligence Planning Systems Conference 1998 (AIPS-98) Planning Competition as a standard way of expressing planning domain problems for the competition. In the following years, the competition is repeated and the language is improved along with it. This language is mainly influenced by STRIPS [10] however it is extended to contain expressiveness of many other planning languages like ADL [28] to incorporate logical propositions and UMCP [8] to incorporate actions. Its syntax is very similar to LISP programming language.

2.2.1 Parts Of The Language

PDDL has two parts; one for specifying the domain and one for specifying the problem. The domain definition contains the whole physical information about the problem space. For example actions, axioms and predicates are defined in this part. The problem definition contains the initial state of the domain and the goal state that is aimed to be reached. These parts of PDDL are given in Figure 2-2 (a) (domain definition) and Figure 2-2 (b) (problem definition).

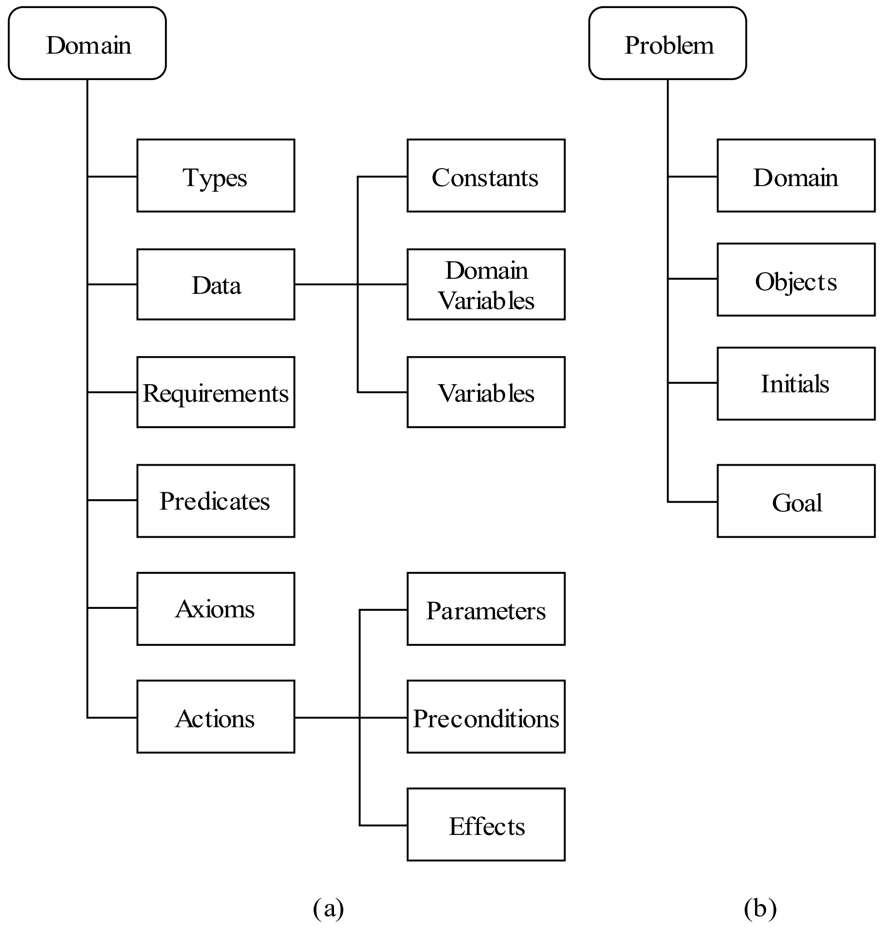

Figure 2-2 PDDL Structure

2.2.2 PDDL Domain

The domain describes the world that the problem is going to be solved in. It captures every aspect of the world to be used by the planning mechanisms. During the problem solving phase, the successive states of the world are generated by the rules defined in this part.

2.2.2.1 Requirements

The requirements are described as a set of flags which specifies the fields that the domain need to cover in order to define the problem space. For example, a domain might cover *strips* and *adl* to use actions and propositional operators and quantifiers in the domain elements.

2.2.2.2 Types

PDDL supports types for parameters of actions, parameters of predicates or variables. All domains intrinsically support *object* and *number* types however they can be extended to better express complex variables.

PDDL supports inheritance of objects and derived types can be used in place of base types as it is in traditional object-oriented languages. PDDL also specifically supports the *fluent* type which is a special variable for the planning domain that changes value from state to state.

2.2.2.3 Data

There are different types of data definition capabilities of the PDDL language. The first of them are *constants*. They define symbolic values of the specified types and they do not change their value as expected. They are visible throughout the scope of the domain. *Domain variables* like constants are scoped over the whole domain but their value can be manipulated. They are like global variables in traditional programming languages. Local *variables* can be defined inside actions, axioms or predicates in a similar fashion to the ones in programming languages and their scope is the place they are defined.

2.2.2.4 Predicates

The usage of *Predicates* is same as they are used in first order logic. Predicates are used to attribute properties to an object or associate a relation to more than one object.

2.2.2.5 Axioms

Similar to the logical formalisms, *axioms* in PDDL domain assert the implications resulting from a satisfied situation. They assert predicates to the domain if the axiom context holds.

2.2.2.6 Actions

Actions are the most important entities of a planning domain. They trigger the state transitions of the domain. In order to reach the goal of a planning problem, *actions* are used to control the succession of situations. If the goal is satisfiable then the solver outputs a sequence of actions that yields the desired result.

Action *parameters* might be used as input during the execution of the action or they can be used to check the *preconditions* of the action or they are used to produce effects related with them. *Preconditions* of an action constitute a set of logical assertions which should be true before that action fires. They are the prerequisite conditions for an action to start. *Effects* are the consequences of an executing action. They cause transition to the current state by changing valuations of variables (fluents) or adding new predicates on objects. Actions alter the current state and in a sense they closely resemble the World-Altering Web Services. Similarly, actions without any effects are similar to Information Collecting Web Services. In Figure 2-3, a PDDL domain example [39] is given to illustrate the domain statements explained in this section.

```
(define (domain carry-balls)
  (:requirements :strips :typing :adl :domain-axioms)

  (:types (robbot gripper ball - object room - location))

  (:predicates (at-robby ?r - robot)
               (at ?b - ball ?r - room)
               (free ?g - gripper)
               (carry ?o - ball ?g - gripper)
               (has ?r - robot ?g - gripper))

  (:action move
    :parameters (?fr - room ?to - room)
    :precondition (at-robby ?fr)
    :effect (and (at-robby ?to) (not (at-robby ?fr))))

  (:action pick
    :parameters (?b - ball ?rm - room
                 ?rb - robot ?g - gripper)
    :precondition (and (available ?rb) (at ?b ?rm)
                       (at-robby ?rm))
    :effect (and (carry ?b ?g) (not (at ?b ?rm))
            (not (free ?g))))

  (:action drop
    :parameters (?b - ball ?r - room ?g - gripper)
    :precondition (and (carry ?b ?g) (at-robby ?r))
    :effect (and (at ?b ?r) (free ?g)
                 (not (carry ?b ?g))))

  (:axiom
    :vars (?r - robot ?g - gripper)
    :context (exists (?g)
             (and (has r g) (free g)))
    :implies (availabe ?r)))
```

Figure 2-3 PDDL Domain Example

In this domain, some actions are defined to carry the balls to the final destination. Actions *pick* and *drop* are used to carry the ball and *move* changes the position of the robot. Finally, the axiom is stating that the robot can only carry a ball if there exits a free gripper on it.

2.2.3 PDDL Problem

The PDDL problem defines the goal of the planning problem at hand. The final state of the domain is described in the *goal* statement. The first order quantification and operators could be used in order to specify the goal. The problem section also defines the initial state of the domain in the *init* statement. Planners should find a partially ordered set of actions that yield the goal state from the initial state. In Figure 2-4, there is an example PDDL problem [39] for the domain example given previously.

```
(define (problem carry-all-balls)
  (:domain carry-balls)

  (:init (room rooma) (room roomb)
         (ball ball1) (ball ball2)
         (gripper left) (gripper right)

         (at-robby rooma) (free left) (free right)
         (at ball1 rooma) (at ball2 rooma))

  (:goal (and (at ball1 roomb) (at ball1 roomb))))
```

Figure 2-4 PDDL Problem Example

This problem is stating that the goal of the robot is to carry the balls from one room to another.

2.3 OWL-S

As the content over web multiplied and the number of services increases day by day, the discovery, integration and execution of resources are becoming an exceedingly overwhelming job if done manually. If this process is automated on the other hand, applications could mine, analyze and understand the content unambiguously, which is not possible unless a standard is homogenously used all over the web. In this extent, a serious effort is going on by the DARPA Agent Markup Language (DAML) Program.

This program is founded to develop a language in order to help the autonomous agents understand the web not just syntactically but also semantically. A series of languages are produced by this group of which the later ones inherit and extend the features of the former ones. The languages, DAML+OIL [5], DAML-S [6] and finally OWL-S [27] are released by this group.

OWL-S, which is specifically developed for services hosted on web, is an ontology that aims to point out the functionalities of services. It is in XML and mainly follows the Resource Definition Framework (RDF) [30] language definitions. The main objectives of this language are listed as follows:

- Automating service discovery
- Automating service invocation
- Automating service composition and hence interoperation

2.3.1 Upper Level Structure of OWL-S

OWL-S offers a model for service definitions. Every service defined in OWL-S should declare *ServiceProfile*, *ServiceGrounding* and *ServiceModel*. These are three different views which look from a different aspect to the same service. *ServiceProfile* describes the service semantically for discovery. *ServiceGrounding* explains the interface of the service for interoperation and *ServiceModel* specify the inner structure of the service for composition. This combined model [27] is depicted in Figure 2-5.

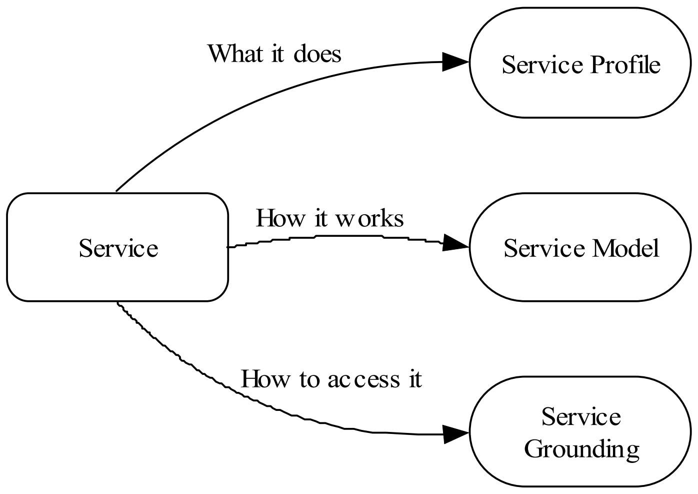

Figure 2-5 OWL-S Model

2.3.2 Service Profile

ServiceProfile contains all the necessary information in order to discover the service. The function of the service is detailed in this modeling element and helps the service seekers locate the service. Attributes of the service profile is given in Figure 2-6.

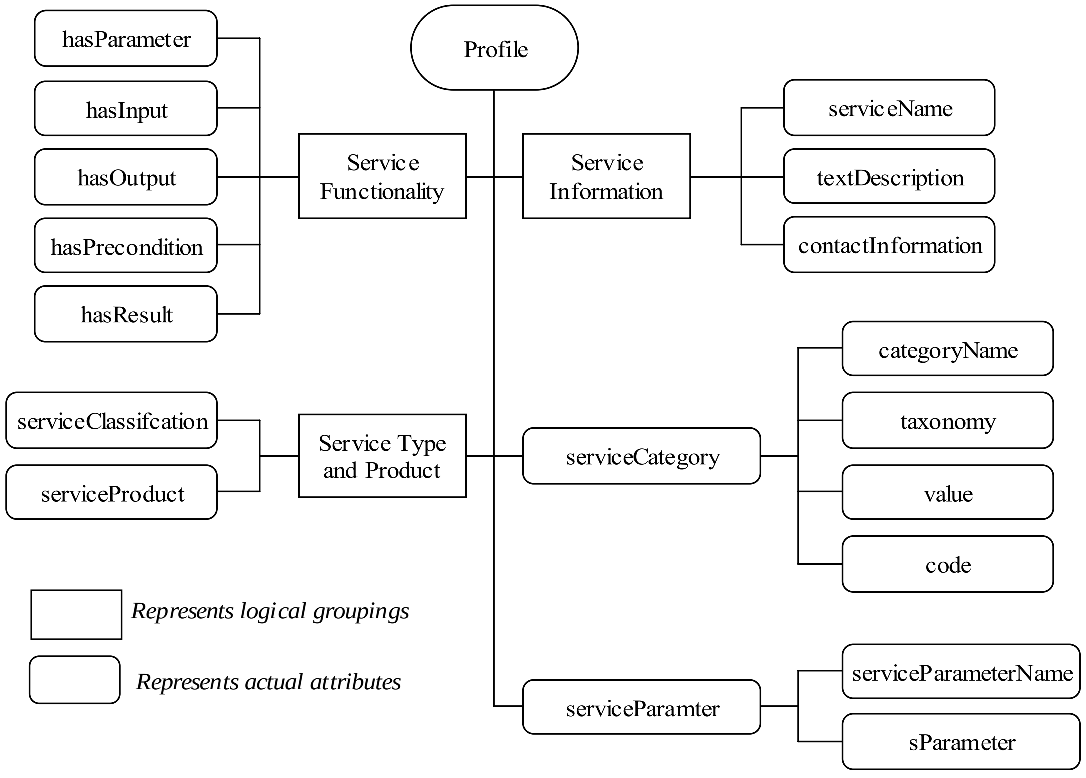

Figure 2-6 OWL-S Service Profile

The profiles represent services in order to specify their purposes. The profile attributes might be grouped under categories according to their purposes.

Under the *Service Information* category, general information about the service is listed. The name of the service, a text based description and the contact person of the service provider is given. The data presented here is for human interpretation and it is format free.

Service Functionality outlines the interface of the service in order to give information about its inputs, outputs, effects and preconditions. All of the attributes listed here, if owned by the service, might be linked with the associated *Service Model Process* instances for a detailed description but there is not any constraint dictated by OWL-S. Service Functionality gives a higher level syntax to describe how the web service will interact with the outer world.

As an attribute, *seviceParameter* is a list of properties that might be given as additional information for the service. Inside the list, there are tuples of names and OWL ontology references. Here the name could be some literal or a link to an actual *Process* parameter and OWL ontology references are used to semantically bind the parameter to an element of ontology tree. Ontology trees are prepared separately and their scope covers a business sector like electronic store sale or flight reservation system.

Another attribute *serviceCategory* is used to give a reference to the categorized information or taxonomies which are used together with the domain of service but outside the scope of OWL-S. This attribute provides an extension point for such external links or interoperability with the already prepared categorizations.

Finally, grouped under *Service Type and Product*, there are two more attributes of the *Profile*. The first attribute *serviceClassification* stores reference to service OWL ontology of the business domain and *serviceProduct* stores reference to product OWL ontology.

2.3.3 Service Model

Service Model describes the inner structure of the services. Each service has a *Process* to represent its internal composition. *Service Model* is influenced by many previous industry standard languages and studies. One of the most important one is the AI planning language PDDL [36]. Also Workflow Process Definition Interface produced by Workflow Management Coalition (http://www.wfmc.org/), a study on situation calculus for complex actions [17] and action based semantic web markup [25] are some other work known to be have affected the *Process* of *Service Model*.

Process tells how the clients will use the service. It describes the inputs, outputs, results, preconditions and effects of the service. Prerequisites and consequences of the service together with the expected inputs and possible outcomes are described in detail to the service requester by the service provider. In that sense it is quite similar to PDDL *action*s. The structure of *Process* is given in Figure 2-7.

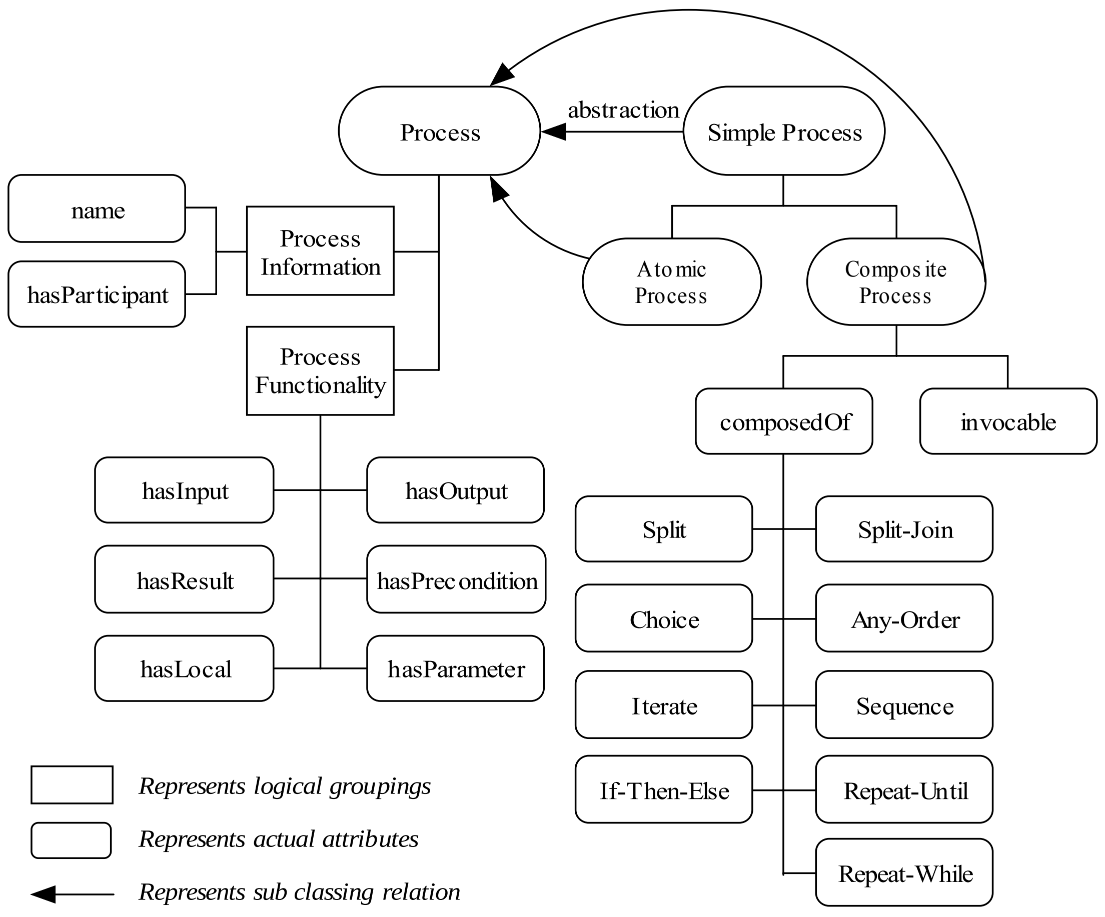

Figure 2-7 OWL-S Service Model

In *Process Information* attribute group, the name and the participants of the *Process* are specified. Normally the default participants are the clients and the server which hosts the service. If additional participants exist they are listed under this parameter.

Process Functionality group, like the corresponding group in *Profile*, details the function of the service. Here the exact interface of the service is given. Intuitively *Inputs* are data given to the service to process. *Outputs* are the information retrieved by the service as a relevant byproduct of its execution. *Inputs*, *outputs* and *local* variables are described as subclass of RDF resource *parameter* class inside their attributes respectively. Every *parameter* class is bound to a data type and might have some restrictions to better describe the ranges that are accepted. *Preconditions*, like actions of PDDL, are the prerequisite logical expressions that have to be true in order to invoke the services. *Results* of the services are used to determine the success and effects of the service.

Process is the base RDF resource class and instantiated by an *Atomic* or a *Composite Process* derived from itself. *Atomic Processes* represent single-step services which are directly executable according to their associated grounding. Composite *Processes* represent set of *Atomic* or *Composite Processes* which are connected to each other with *Control Constructs*. *Split*, *Split*-Join, *Sequence*, *Choice*, *If-Then-Else*, *Iterate*, *Repeat-While*, *Repeat-Until* and *Any-Order*, which are subclasses of *Control Construct* RDF resource class, are used to define the composition flow. These constructs are defined to be exactly as they are in common workflow definitions [12].

Finally, *Simple Processes* are abstractions or simplified views of *Atomic* and *Composite Processes*. They are not executable and not bound to *grounding* but perceived as a single-step process, like *Atomic Process*, which eases the planning and reasoning tasks over processes.

2.3.4 Service Grounding

Grounding describes access details which are: the used protocol, the message format, the transport and addressing of services. It is a mapping from abstract definition of the service to a concrete specification.

WSDL [42] which is an industry standard for Web Services communication is used for binding abstract descriptions with concrete definitions. In WSDL *binding* of services exactly serves as *grounding* in OWL-S.

Authors of OWL-S claim that both languages fit each other perfectly [27]. WSDL is weak in abstract definitions of services in semantic, schematic and taxonomic aspects, on the other hand OWL-S has no support for concrete communication in any standard form of industry protocols like SOAP or HTTP.

Atomic Processes, which are the executable entities of OWL-S, are bound to WSDL *operations* to concretely specify the abstract Web Service interface. Table 2-3 summarizes the correspondences between both languages and thus outlines *grounding* of OWL-S through WSDL.

Table 2-3 WSDL and OWL-S Correspondences

OWL-S	**WSDL**
Atomic Process	Operation
Atomic Process with input and then output	Request-response operation
Atomic Process with input only	One-way operation
Atomic Process with output only	Notification operation
Atomic Process with output and then input	Solicit-response operation
Atomic Process inputs	Operation input message
Atomic Process outputs	Operation output message
Atomic Process input and output types	Abstract types

WsdlGrounding class is used to bind *Processes* to their associated WSDL description inside OWL-S. It contains a list of *WsdlAtomicProcessGrounding* instances which connects Atomic Processes to service *operations* in WSDL. This structure is given in Figure 2-8.

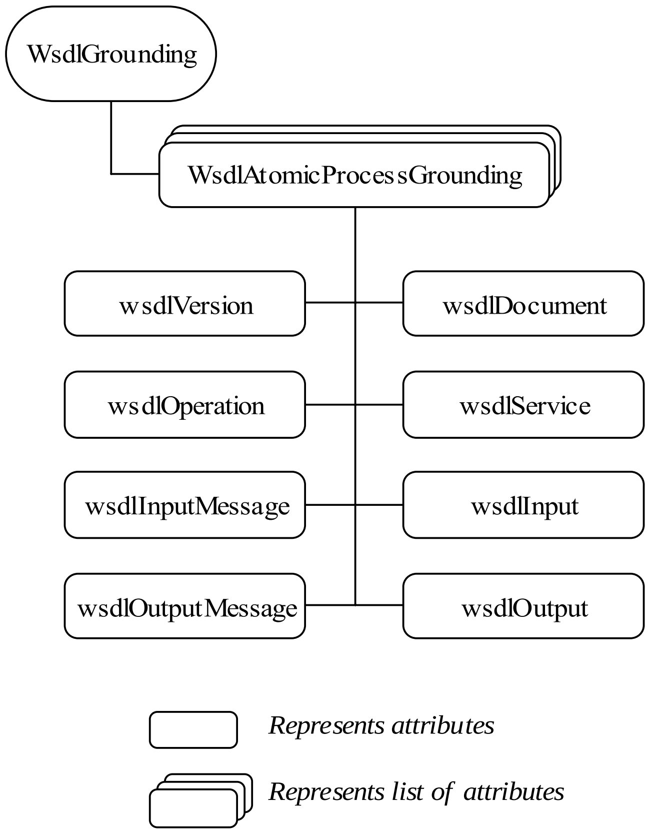

Figure 2-8 OWL-S Service Grounding

WsdlAutomicProcessGrounding instance contains referencing information for its associated WSDL grounding. The version of WSDL in use and WSDL document are referenced using this attribute. The associated WSDL *operation* is given in *wsdlOperation* attribute. Finally, the rest of the attributes refer to WSDL *operation inputs/outputs* and WSDL *operation input/output messages*.

2.4 Automatic Web Service Composition

Web service composition is finding an integrated schema of atomic Web Service operations such that when executed, the desired effect is produced. Web services and other sub-schemas are the basic building elements of the schema however extra logical or imperative operators control the data flow and the execution order.

Schemas are domain specific most of the time and they are meaningful only in the solution of the automated composition technique. However languages like OWL-S with their flow definition constructs are the efforts for standardizing the resulting schemas.

Most of the time a composition is generated after a specific problem is encountered. The resulting composition takes the inputs of the problem and generates the expected outputs and effects after it is executed. In this aspect a composition highly resembles to a higher level function which is composed of primitive or lower level functions inside.

2.4.1 Rationale for Automating Web Service Composition

As the enterprise applications getting more and more complex nowadays, some of the subtasks are to be distributed to the partner companies which are specialized in their field since customer demands are too complex to deal with as a whole.

For example, a traveling company should utilize different flight and hotel reservation systems to arrange the entire scheduling. If the company concentrates on several cities and several air companies, then it would be reasonable to build a system that incorporates with selected companies at once and then customize the queries based on the user demands at runtime. But if the capability is extended to realize travel arrangements throughout the whole world then it is an unduly difficult job to integrate all air companies and hotels all around the world. Yet even though such an effort was spent, application developers should continuously integrate the newly added Web Services, remove the closed ones, and update the integration of the ones that have changed their interface which are equally difficult and manually almost impossible tasks. For that reason, in order to build such an application, the selection and integration of heterogeneous Web Services should be done at runtime after collecting the user demands.

Several examples are posed to be solved with the help of Automated Composition Techniques and in fact they are the stimulus of the problem at hand, since not much is done in practical application areas. Some of the best known examples are:

- *Traveling Domain*: It is the domain of trip planning systems that offer to query and book transportation and accommodation according to user-defined constraints [11]. A typical problem of this domain is to plan a trip for a conference attendance with constraints like the date and place of the conference, preferences for certain hotels or airlines [24].
- *Appointment Scheduling Domain*: It is the domain of schedule organizing systems that offer multiple appointments according to user constraints. A typical instance is arranging a schedule after a visit to a doctor that involves tasks of prescription filling in

pharmacy, diagnostic tests in different medical test centers and a final follow-up meeting with the doctor [2].

- *Commercial Sale Domain*: It is the domain of electronic sale system that offers purchasing of items according to customer constraints or quality of service (QoS) [40] parameters. For instance a customer wants to buy a microprocessor but he or she does not want to know where or how to buy the item [21].

2.4.2 Stages of Automated Web Service Composition

As it is discussed in CHAPTER I, there are common stages in the Automated Web Service Composition problem. These stages continuously follow each other as depicted in Figure 2-9.

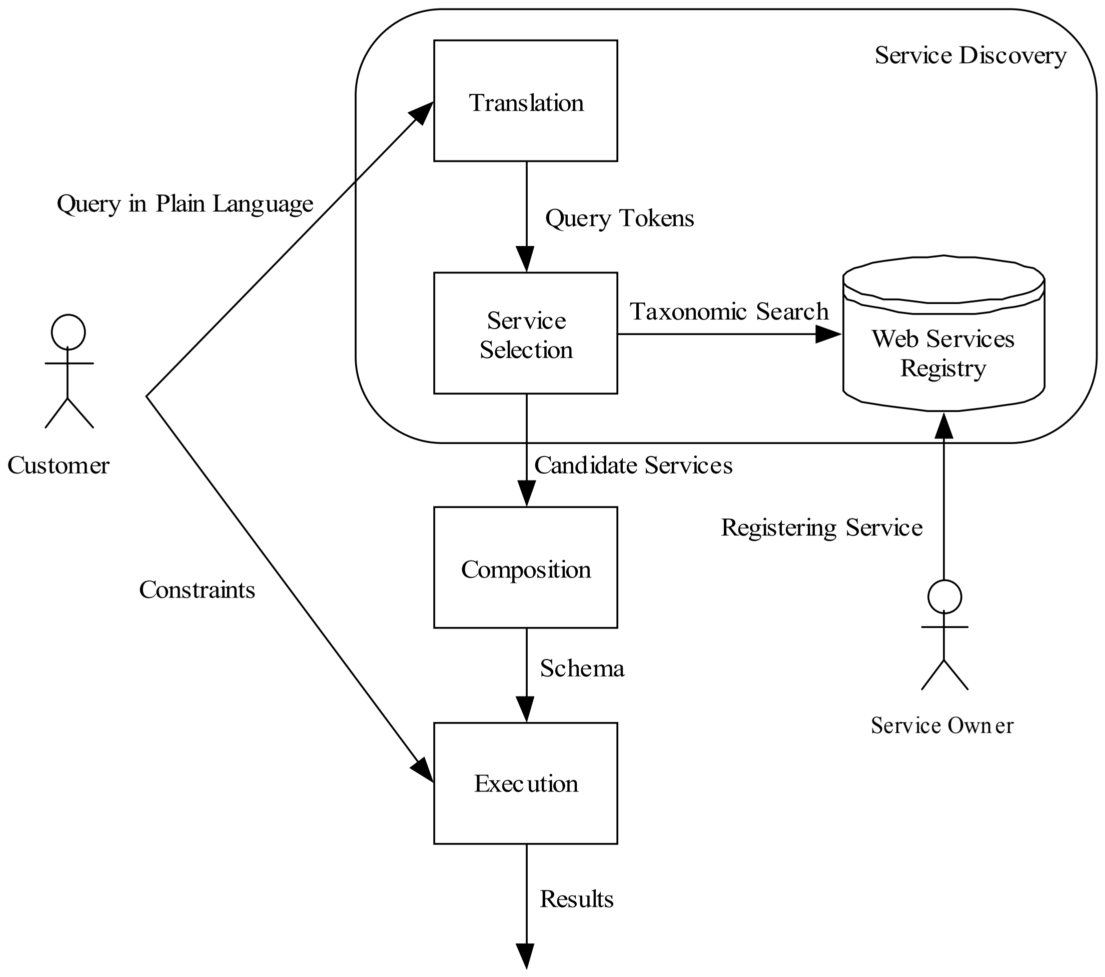

Figure 2-9 Flow of Automatic Web Service Composition

In this figure, the abstract stages of Automatic Web Service Composition are given. In the first step, user requests are translated into machine understandable tokens. These tokens are searched in UDDI or ebXML registries. During the search, taxonomic matches might be employed to enrich the set of candidate Web Services for the composition. Moreover if the ontological specifications of the Web Services are available then the matches returned from registry searches contain more probable candidates for the compositions. The details of the translation of user queries to machine interpretable tokens are generally omitted in scope of this specific problem in the literature since this process mostly falls in the area of Natural Language Processing. Most of the time queries are assumed to be captured in a formal list of actions [26].

In the second stage compositions or schemas are generated from the atomic services that are discovered in the first stage. The composition system usually takes the functionalities of services as input, and outputs the process model that describes the composite service [35]. Finally, the generated schema contains the atomic services as well as the data flow and control logic between them. After the composition, an optional evaluation step is used to select the optimum composition that is the best match to user constraints if more than one composition is generated [26].

In the final stage the schema is executed with the customer inputs, and the outputs are generated. The execution of a schema is not much different from the execution of a programming language procedure since the control logic and data flow is already explicitly defined.

The current literature is more focused on the service composition problem and the execution stages than the discovery of them even tough new studies, like OWL-S, have also aimed to improve the service selection qualities [27, 38, 7]. Many techniques have been applied for the automatic composition of Web Services which can be grouped under two different categories: rule based composition and AI planning based composition. Some of the realized instances of these approaches are explained in the subsequent sections in order to facilitate the comparison of our technique with these existing techniques.

2.4.3 Rule Based Composition

Rule based composition is a technique to solve the relations between services using a static set of rules to check whether two services can be composable [26, 34]. These rules compare the specific properties of two services according to their rules and if they match they are added to the composition. Starting from the input specification, the composition is extended successively to contain matching services until the goal specification is added to the service. Finally the generated composition is a tree which interconnects the matching services.

2.4.3.1 Composability Rules Approach

A proposed technique [26] following the rule-based composition approach uses the *composability* rules to find out the relations between services. These rules compare the syntactic and semantic features of Web services to determine whether two services are composable [26]. The composition request is taken in a language called CSSL (Composite Service Specification Language) developed in the scope of their study. It is very similar to WSDL but contains some ontological extensions in order to utilize taxonomic matches between services.

After the request is taken, a composition is generated iteratively by applying the composability rules [26] in a chain as shown in Figure 2-10.

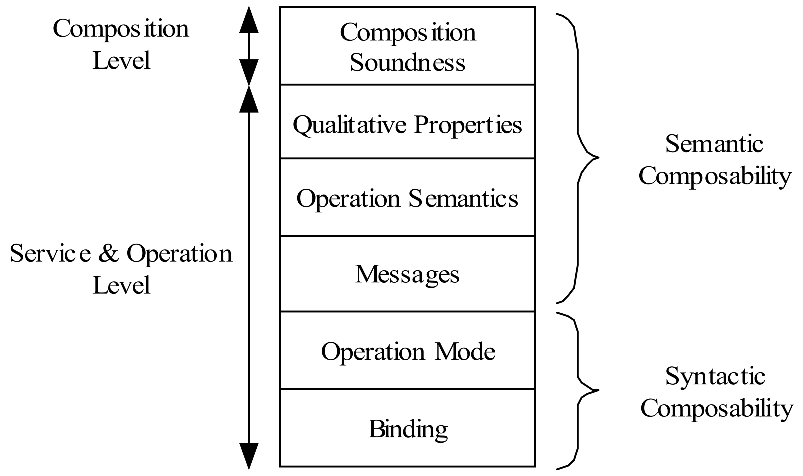

Figure 2-10 Composability Model for Web Services

Syntactic Rules

Binding specification rule controls the protocol of the web services (eg: SOAP, HTTP Get/Post, MIME). One operation is binding compatible with another one if the comprising web services have compatible bindings. Since no translation from one type to another is provided, operations should be protocol compatible in order to expect a valid communication between two services.

Operation Mode specification controls the data flow of the operation. For example, if the service operation in hand is a *Notification* operation (operation with output only) then the matching operation should be *One-way* (operation with input only) in order to decide that their operation modes are matching.

Semantic Rules

Message specification rule checks the operation parameters (messages as in WSDL) for data type agreement. Two operations are message composable if for all input messages of one, there is an output message which is *data type compatible* with the other and their role type (eg: stock value, price), and unit specifications (eg: dollars, kilometers) are taxonomically matching. A message *M* is *data type compatible* with a message *M´* if every parameter (as *part* in WSDL) of *M* is *directly* or *indirectly compatible* with a parameter of *M´* [26]. A message part is *directly compatible* with another one if their data type is exactly the same. A message part *P* is *indirectly compatible* with another part *P´* if *P* is derived from P´.

Operational semantic rules check the *domain*, *synonym domains* and *specialization* descriptions of the Web Service operations. Each Web Service operation is assumed to have these three attributes. *Domain* specifies the scope that the operation is meaningful and *synonym domains* are domains which are known to be equivalent to the *domain* of the operation. Finally *specialization* describes the characteristics set of the operations. Two operations are composable according to the operational semantics if the *domain* of one is equal to the domain of the other or it is a member of *synonym domains* of the other or the intersection of *synonym domains* of both operations is not an empty set. Also the *specialization* of one operation should be a subset of the other.

Qualitative properties of services define the *fee*, *security* and *privacy* attributes of Web Service operations. A service operation *O* is composable with another operation *O´* if price of *O* is more than *O´* and their security attributes are same (eg: if one of them is known to be secured, the other one should be secured as well) and privacy policy of *O* should a subset of *O´*.

This rule asserts that if an operation is going to be added to the composition it should be reliable as the integrated one and cheaper than it.

The final composability rule checks the whole composition instead of service operation couples using the *category* attribute of service operations. The *Category* of operations describes the business area (eg: car dealer, flight reservation) that the operation serves. The resulting composition is compared with a library that contains templates. Compositions are treated as directed graphs where the directed edges show the data flow direction and vertices denote the atomic web service operations. Templates are also directed graphs where the vertices are replaced by *category* attributes of the operations. They define a generic flow of job in certain business fields. In Figure 2-11 such an example is given. The graph on Figure 2-11 (a) is a template graph and the one on Figure 2-11 (b) is extracted from the composition. Since the extracted composition graph is a sub-graph of the template graph it is accepted to be sound.

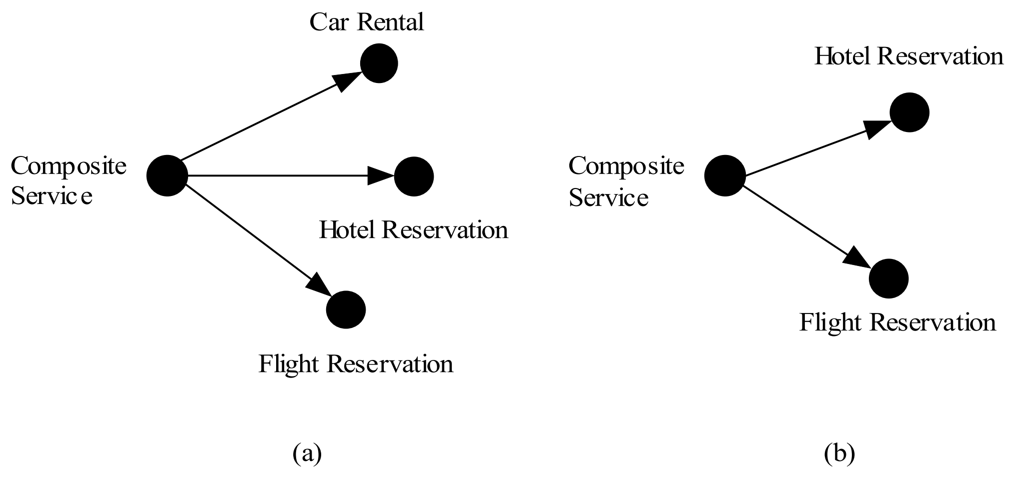

Figure 2-11 Composition Soundness Graphs

Quality of Service (QoS) Selection

After the *composition rules* are applied if more than one composition remains they are exposed to a selection phase. The selection is performed according to three QoS parameters; *composition ranking*, *composition completeness* and *composition relevance*. *Composition ranking* is the occurrence of the current composition as a sub-graph of templates. *Composition completeness* is a ratio that defines how tight the composability rules are followed. If this ratio is low it is claimed that algorithm might return plans in which most of the composite service operations are not composable with component service operations [26]. Finally *composition relevance* compares the composition at hand with templates according to the number of matching edges. The best composition is selected according to these QoS parameters and returned as the resulting composition.

2.4.4 Planning Based Composition

For the solution of Automated Web Service Composition problem, most actively used technique is the AI planning technique. Most apparent reason behind this preference is the great similarities between these two fields.

Both the planning problem and composition problem seek a (possibly partially) ordered set of operations that would lead to the goal starting from an initial state (or situation). Operations of the planning domain are actions (or events) and operations of the composition domain are the Web Services (or Web Service operations to be more precise). Web services have preconditions and effects just like actions and this makes planning domain solutions attractive for the composition problem. Since Web Services induce effects and alter the current state, composition space can be viewed as a state space where the executing services trigger state transitions.

Viewing composition problem as an AI planning problem, different planners are employed for the solution and the ones that are most relevant to this thesis are explained briefly in the following sections.

2.4.4.1 Estimated Regression Planning Approach

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 [21]. Relaxation is used to neglect the constructive or destructive effects of actions to each other. This reduces the number of interactions between them allowing the construction of full state space which is called the *regression graph*. This graph actually shows the minimum sequence of actions required to achieve each sub-goal of the goal and used as a basis for the heuristic for forward planning.

In this approach, the composition problem is seen as a PDDL planning problem and efforts are condensed to solve the problem in PDDL domain referring to the common difficulties of Web Services domain. In fact, a translator [22] has been written which converts DAML-S (an earlier version of OWL-S) and PDDL into each other. This shows that the composition problem can be (informally) reduced to a planning problem and in that sense working in PDDL domain is not much different indeed.

The estimated regression solver considers Web Services 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.

One of the important argued points in [21] for the composition problem is that using a Prolog like solver mechanism, plans cannot be constructed effectively since backtracking mechanisms are not suitable for the composition problem domain. During the composition or plan generation, the obtained information in prior steps are propagated as inputs or constraints to the next steps and backtracking inefficiently prunes the composition by resetting the already collected and verified information which is not related to the failed condition that has triggered backtracking. In order to avoid this problem in PDDL, domain instead of representing it as an effect, an extra field, *value*, is added to *actions*, which represents the information received after executing the *action* (invoking the Web Service).

Another important issue in [21] is about knowledge representation. During planning, terms like “quote of stock” or “room rate of hotel” are *learned* terms and they should be treated differently from other variables. Since their values are acquired during execution, applying the closed world assumption (CWA) is incorrect since their value might have not been learned yet or their value has been changed. For that reason *learnable* terms are surrounded with a meta-level predicate *know-val* which controls whether the value is known or not without leaving it to CWA.

2.4.4.2 Hierarchical Task Network Planning Approach

Hierarchical Task Network (HTN) planning is an AI methodology which makes planning by decomposing the tasks into subtasks until primitive tasks are reached. As the name of the methodology implies, planning process operates on task networks which is a collection of tasks and task order constraints, tasks preconditions and task effects. In that sense, the primitive tasks resemble the actions of the planning domain. The most apparent difference is that in planning with actions, the planner tries to reach the goal whereas in HTN planning the aim is to find a decomposition set which contains the primitive actions that are consistent with each other.

This planning technique has been applied to the composition problem to develop software to automatically manipulate DAML-S (former version of OWL-S) process definitions and find a collection of atomic processes that achieve the task [43]. HTN is claimed to be suitable for automatic transformation of DAML-S descriptions since the task decomposition process of HTN closely resembles the composite processes in DAML-S. For that reason SHOP2, an HTN planner, is used in [43] to generate plans in the order of its execution. Preserving the execution order, SHOP2 is able to process preconditions of actions, which contain external Web Service calls, in the order they should be resolved.

SHOP2 Constructs

SHOP2, like most of other HTN planners, has *operators* and *methods* to do the task decomposition. *Operators* are composed of four elements; *primitive task* with a number of input parameters, *preconditions*, *add* and *delete list* which contain the literals becoming true and false respectively after the execution. In fact, *add* and *delete list* are, when combined, defines the effect of the operator. *Operators* can be expressed as a fixed tuple of five elements:

$$O = \langle H, V, P, A, D \rangle \qquad (2\text{-}1)$$

Here, H, V, P, A and D denote primitive task, parameter list, preconditions, add list and delete list respectively.

Methods are defined as a *compound task* with a number of input parameters, *preconditions* and partially ordered *task list*. This *task list* can contain further *methods* and *operators*. *Methods* can be expressed as a four-tuple:

$$M = \langle C, V, P, T \rangle \qquad (2\text{-}2)$$

Here *C*, *V*, *P* and *T* denote *compound task*, *parameter list*, *preconditions* and *task list* respectively. Moreover, inside methods, more than one *precondition-task list* pair can be given. In this case, the *task list* with a satisfied *precondition* is chosen to be executed. It is shown as a variable-dimensioned tuple:

$$M = <C, V, P1, T_1, P_2, T_2, \ldots, P_N, T_N> \qquad (2\text{-}3)$$

DAML-S to SHOP2 Translation

DAML-S Composite Processes are translated into *operators* and *methods* with an algorithm which generates a *method* or *method set* for each construct that appear in the *Composite Process* of DAML-S. *Atomic Processes* with output, *Atomic Process* with effect, *Simple Process*, *Sequence*, *If-Then-Else*, *Repeat-While*, *Repeat-Until*, *Choice* and *Any-Order* (Unordered) *Control Constructs* of DAML-S are translated to *methods* of SHOP2.

As an example, *DAML-S If-Then-Else Process*, *Q*, is converted to a method as:

$$M_Q = <C_Q, V_Q, P_Q \, U \, \{\pi_Q\}, T_{Q\text{-}If}, P_Q, T_{Q\text{-}Else}> \qquad (2\text{-}4)$$

In this representation π_Q, $T_{Q\text{-}If}$ and $T_{Q\text{-}Else}$ denotes condition of *Q*, *if-tasks* of *Q* and *else-tasks* of *Q* respectively. Here the trick is to add a conditional to *preconditions* of *if-tasks* and use *preconditions* without adding the conditional for *else-tasks*.

There are three shortcomings of the translation process. Firstly, A*tomic Processes* with both output and effects cannot be processed. Secondly, conditional effects cannot be handled. Finally *Split* and *Split-Join Control Constructs* cannot be translated since there is no support of concurrency in SHOP2.

2.4.4.3 GOLOG Approach

Another solution for the composition problem is published in which situation calculus [25, 24] is employed. Situation Calculus is a first order logical system of actions and situations which is first appeared in [18] and described in detail in [29].

Situation Calculus

In situation calculus, dynamically changing world is being modeled. Actions affect the representation of domain and alter the current state. In other words, domain knowledge or observable facts of the world change after the execution of actions.

For example, if there is a turkey in the world and if a pointed gun is fired then the turkey dies (from famous Yale Shooting Scenario). Here the initial situation is that there is a turkey which is alive and there is a loaded gun pointed at turkey and this is our initial knowledge. Action is firing the gun. Effect of firing a loaded gun which is pointed out to a living turkey is that turkey is not alive anymore. Effects of the action altered our knowledge since in this new state turkey is now dead.

When an action is executed a situation transition occurs. In that sense situations are in fact history of actions. Knowledge that we have is referenced to the situations. The initial situation is described as S_0. Performing an action in a situation is represented with a function which is from A x S to S, called *do*, where A and S denotes the set of actions and the set of situations respectively. For example, the situation transition of shooting example is expressed as:

$$S_{dead} = do(fire(Gun), S_0) \qquad (2\text{-}5)$$

Actions affect certain entities and certain attributes of the world. These are captured in fluents of the world. Fluents, like variables, get different valuations as a result of effecting actions. For instance, since we are interested in the health of turkey in the shooting domain *alive* fluent could be used. Then in the initial situation we can say $alive(Turkey, S_0)$ and after the firing action turkey dies changing the value of the fluent; $\neg alive(Turkey, S_{dead})$.

Actions might have preconditions which control whether they can be executed or not. For example, in order to fire the gun we should have a loaded gun. Preconditions of actions are represented by a *Poss* (ible) predicate whose parameters are an action and a situation. Situation calculus solver permits the execution of an action if it is possible to do so. Continuing with our example it is expressed as:

$$\forall s\ Poss(fire(Gun), s) \leftrightarrow loaded(Gun, s) \tag{2-6}$$

It states that firing the gun is possible if and only if the gun is loaded for all situations of the world. Universal quantifier can be omitted assuming unbounded free variables are universally quantified. Here completion semantics is required to have the model complete and rule out the possibility that another situation might enable the firing of gun creating another model which is not intended.

Finally the effects of actions are represented with successor state axioms. They define the conditions under which the valuations of the fluents change when affecting actions occur. For our example they are:

$$Poss(fire(Gun), s) \wedge alive(Turkey, s) \supset \neg alive(Turkey, do(fire(Gun), s)) \tag{2-7}$$

$$Poss(fire(Gun), s) \wedge loaded(Gun, s) \supset \neg loaded(Gun, do(fire(Gun), s)) \tag{2-8}$$

$$Poss(load(Gun), s) \wedge \neg loaded(Gun, s) \supset$$

$$loaded(Gun, do(load(Gun), s)) \qquad (2\text{-}9)$$

Here in these successor state axioms it is said that firing gun unloads the gun and kills the turkey. Again here the completion semantics are required to avoid the frame problem [18]. Also unique names axiom (UNA) is added for the actions to avoid multiple models satisfying the same domain violating the soundness of formalism.

GOLOG

GOLOG [17] is a situation calculus implementation which extends the basic formalism with complex actions. The definition of complex actions is recursive since in addition to the primitive actions, they can contain other complex actions including themselves. Contained actions are connected with extralogical operators.

GOLOG solver that has been implemented in Prolog uses a deductive theorem prover, *Do*(*A, S, S´*), to solve actions according to the current situation where *A* denotes an action, *S* denotes the current situation and *S´* denotes the next situation. *Do* projects action to the current situation and obtains the next situation. If the action is primitive and if its preconditions are satisfied (or *Poss*-ible) then it is executed. If the action is not primitive, it is first expanded and then solved according to the axioms of its extralogical constructs. In Table 2-4, axioms of these extralogical constructs are given.

Table 2-4 Constructs of GOLOG

Meaning	Definition
Primitive Actions	$Do(A, S, S') \equiv Poss(A, S) \wedge S' = do(A, S)$
Test Actions	$Do(\Phi?, S, S') \equiv \Phi \wedge S' = S$
Sequence of Actions	$Do([\delta_1 ; \delta_2], S, S') \equiv Do(\delta_1, S, S'') \wedge Do(\delta_2, S'', S')$
Nondeterministic Choice of Actions	$Do([\delta_1 \mid \delta_2], S, S') \equiv Do(\delta_1, S, S') \vee Do(\delta_2, S, S')$
Nondeterministic Choice of Arguments	$Do((\pi x)\delta(x), S, S') \equiv (\exists x)\, Do(\delta(x), S, S')$
Nondeterministic Iteration	$Do(\delta^*, S, S') \equiv Do([] \mid [\delta ; \delta^*], S, S')$

Complex actions are expanded until primitive actions are obtained. In fact, the combination of these constructs defines the body of the complex actions. The head of complex action is defined as a procedure which can have parameters. As an example, *ancestor* procedure and its body are given:

```
proc ancestor(X, Y) [
        parent(X, Y)? |
        (πZ) [parent(X, Z) ; ancestor(Z, Y) ]                    (2-10)
```

The procedure *ancestor* is using recursive call to itself in the scope of a projection operator (nondeterministic choice of argument) to find the ancestor.

Procedures are like macros [17] of programming languages which are substituted with their definition and then executed. They resemble to HTN *methods* however procedures do not have any preconditions like *methods* and their effect is generated by the execution of their sub-actions.

Web Service Composition with GOLOG

Being a suitable logic programming language, GOLOG is used as a tool for web service composition problem in [24]. In this work, web service composition problem is assumed to be the execution of generic compositions with customizable user constraints.

In this method, a generic GOLOG procedure (complex action) is written as a solution of a specific composition problem. It is said to be generic since it cannot be executable without user constraints. After the user specifies the constraints, it is executed and solver tries to generate the aimed effect according to the runtime behaviors of the services. The output of this method is a running application which satisfies the user requests, instead of generating a plan which could possibly[1] do so. The main aim of the study is to show that GOLOG, as a logical programming language, can be used effectively for solving the composition problems manually.

Another construct different from the regular ones is added to the language in order to express the procedures of service compositions more effectively. *Order* construct (shown with ":" operator) is used to combine the list of sub-actions like the *sequence* construct however the difference is that if the sequence of actions cannot be achieved then a series of actions which allow finishing the execution of the sequence is searched and executed. For example, assume that there is a complex action defined as [$A_X ; A_Y$] then if $Poss(A_Y, do(A_X, S))$ fails then the execution fails. However if they are defined as [$A_X : A_Y$] (with colon) then if it is not possible to execute A_Y, the solver tries to find a sequence of actions that it could execute A_Y afterwards. That is to say, the planner tries $Poss(A_Y, do(A_N, \ldots (do(A_2, do(A_1, do(A_X, S)))) \ldots)$ to

[1] Possibly is used since plans do not guarantee the desired effect when executed.

find a series of actions with an execution as [A_1 ; A_2 ; … ; A_N] between A_X and A_Y.

Since the actions that can be inserted between two specified actions of *order* construct are determined during runtime according to the current situation, in a sense it is nondeterministic like some other constructs of the language. On the other hand it improves the success possibility of the procedure. Therefore the generic procedure is not executed solely but a limited search is applied during execution. This construct is built with present constructs as [24]:

$$A_X : A_Y \equiv A_X \, ; \, [\neg Poss'(A_Y)? \, ; \, (\pi A)[Poss'(A)? \, ; \, A]\,]^* \, ; \; Poss'(A_Y)?] \qquad (2\text{-}11)$$

$$A_X : A_Y \equiv A_X \, ; \, \textbf{while} \; (\neg Poss'(A_Y)) \; \textbf{do} \; (\pi A)[Poss'(A)? \, ; \, A] \; end \, ; \, A_Y \qquad (2\text{-}12)$$

The second version is given as a more readable version [17]. The equivalent of *order* construct is substituted by the solver in *Do*. The predicate which is used inside the complex action is evaluated by *Do* against the current situation automatically and for that reason it does not contain a situation argument as normal *Poss*.

Another extension to GOLOG is introduced to incorporate user constraints to the programming language. Another predicate *Desirable* is added as a precondition for actions like the predicate to evaluate the user preferences. If an action is going to be executed it should be possible and desired for the current state and *Do* for the primitive actions is modified as follows:

$$Do(A, S, S') \equiv Poss(A, S) \wedge Desirable(A, S) \wedge S' = do(A, S) \qquad (2\text{-}13)$$

Desirable for each action is compiled and added to the program before the execution. If nothing is desired for the action then *Desirable*(*A, S*) is assumed to be true. Compilation of user constraints into a single *Desirable* statement is automatically performed by the application.

As an example, assume that a user wants to reserve a room from a hotel which is by the sea or which has a pool and the hotel should be in Antalya. Here *Desirable* of *reserve* action is compiled into:

$$Desirable(reserve(Hotel), S) \equiv (byTheSea(Hotel) \vee hasPool(Hotel)) \wedge at(Hotel, Antalya) \quad (2\text{-}14)$$

With two assumptions, it is proved that compositions written as GOLOG procedures will eventually terminate in a consistent situation with respect to the theory in [24]. The first assumption is that non-knowledge preconditions of actions that retrieve data from external Web Service calls are true in the theory (initial state and axioms) of the problem. The second assumption says that the evaluated preconditions keep their valuations while the dependent decisions are being made.

CHAPTER III

EVENT CALCULUS

3.1 Basic Formalism

Event Calculus [16] is another logical formalism which resembles situation calculus in basic theory. Likewise it is used with domains where actions, from now on we call events, dynamically affect and change the world. Basically, the formalism can be used for three different purposes:

- Execution through deduction and finding a final situation for theory that is composed of the Event Calculus rules, axioms and initial state. This process is known as temporal projection. This is the same behavior that GOLOG provides.
- Finding a series of events, called narrative, through abduction that satisfies the theory and a goal state (or a partial goal state). In fact, finding a (partially) ordered event set that causes goal to be satisfied is simply planning.
- Finding the theory itself through induction using the Event Calculus axioms, narrative and effected fluents.

3.1.1 Event Calculus Predicates

Formulation of the Event Calculus is defined in first order predicate calculus like situation calculus. Likewise, there are actions and effected fluents. Fluents are changing their valuations according to effect axioms (successor state axioms) defined in the theory of the problem domain.

However there are also big differences between both formalisms. The most important one is that in the Event Calculus, narratives and fluent valuations are relative to time points instead of successive situations. The most appearing advantage of this approach is the inherent support for concurrent events. Events occurring in overlapping time intervals can be deduced. Consequently the narrative does not have to contain the information about order of events when the order is not important or any order is acceptable.

In order to define the theory of a specific problem domain some predicates are used as in situation calculus. These predicates are then used by the solver to produce the desired outcome of the formalism. Table 3-1 summarizes these predicates and their meanings.

Table 3-1 Event Calculus Predicates

Predicate	Meaning
$T_1 < T_2$	Time point T_1 precedes time point T_2 in time line.
Happens(E, T_1, T_2)	Event E happens during time frame between T_1 and T_2 where $T_1 < T_2$.
Happens(E, T)	Instantaneous Event E happened at T defined as *Happens*(E, T, T).
HoldsAt(F, T)	Fluent F holds at time T.
Initially(F)	*HoldsAt*(F, T_0) where T_0 is the time of the initial state.
Initiates(E, F, T)	*HoldsAt*(F, T) when *Happens*(E, T, T_A).
Terminates(E, F, T)	*HoldsAt*($\neg F, T$) when *Happens*(E, T, T_A).
Releases(E, F, T)	Valuation of Fluent F is released, removing the inertial constraints on it.
Clipped(T_1, F, T_2)	*HoldsAt*($\neg F, T$) where $T_1 < T < T_2$
Declipped(T_1, F, T_2)	*HoldsAt*($\neg, T$) where $T_1 < T < T_2$

For the predicate, it is specified that it clears the inertial constraints on the fluent. Inertia [13] is an assumption, which accounts a solution to the frame problem together with other techniques and it is saying that a fluent preserves its valuation unless an event specified to effect (directly or indirectly) the fluent occurs. In other words, a fluent preserves its valuation unless there is an evidence to believe otherwise.

3.1.2 Event Calculus Axioms

Each Event Calculus theory is composed of axioms. There are default axioms of the formalism and others are specified by the author who formalizes the problem domain in the Event Calculus.

In its most primitive use the initial state and the effect axioms for each fluent-event pair is defined inside the theory. Whenever the default axioms of the Event Calculus and the unique names axioms are added, full theory is constructed. Axioms of the Event Calculus are taken from [31] and are given below.

The axioms that define clipping and declipping of fluents are given as follows. The predicate defines a time frame for a fluent that is overlapping with the time frame of an event which terminates or releases this fluent. Similarly *Declipped* defines a time frame for a fluent which overlaps with the time frame of an event that initiates or releases this fluent.

$$\begin{aligned}&Clipped(T_1, F, T_4) \leftrightarrow (\exists E, T_2, T_3)\ [\ Happens(E, T_2, T_3) \wedge \\ &\quad [Terminates(E, F, T_2) \vee Releases(E, F, T_2)] \wedge \\ &\quad T_1 < T_3 \wedge T_2 < T_4\ ]\end{aligned} \tag{3-1}$$

$$\begin{aligned}&Declipped(T_1, F, T_4) \leftrightarrow (\exists E, T_2, T_3)\ [\ Happens(F, T_2, T_3) \wedge \\ &\quad [Initiates(E, F, T_2) \vee Releases(E, F, T_2)] \wedge \\ &\quad T_1 < T_3 \wedge T_2 < T_4\ ]\end{aligned} \tag{3-2}$$

The axioms that define whether a fluent holds till the initial state are as follows.

$$HoldsAt(F, T) \leftarrow Initially(F) \wedge \neg Clipped(T_0, F, T) \tag{3-3}$$

$$HoldsAt(\neg F, T) \leftarrow Initially(\neg F) \wedge \neg Declipped(T_0, F, T) \tag{3-4}$$

Axioms below are the ones that deduce whether a fluent holds or not at a specific time.

$$\begin{aligned}&HoldsAt(F, T) \leftarrow Happens(E, T_1, T_2) \wedge Initiates(E, F, T_1) \wedge \\ &\quad \neg Clipped(T_1, F, T) \wedge T_2 < T\end{aligned} \tag{3-5}$$

$$\begin{aligned}&HoldsAt(\neg F, T) \leftarrow Happens(E, T_1, T_2) \wedge Terminates(E, F, T_1) \wedge \\ &\quad \neg Declipped(T_1, F, T) \wedge T_2 < T\end{aligned} \tag{3-6}$$

3.1.3 Formalization of the Theory

In order to avoid the frame problem, circumscription is used in the Event Calculus. Circumscription is the minimization of predicates of the model that describes the theory. It is defined in the second order logic and the minimum models for the theory are generated for the specified predicates. In the Event Calculus minimization is achieved with the completions of the axioms.

Circumscription is used for the predicate *Happens*. Its definitions are completed in order to avoid any other occurrences of events other than the intended one in other models of the theory.

Similarly circumscription is applied on *Initiates* and *Terminates* axioms as well to avoid any other unexpected effects of the events in other models of the theory. Circumscription on these axioms brings inertia since there is no other effect which could alter the fluents.

Finally the unique names axiom is used for the names of events in order to avoid multiple conflicting models which violate the soundness of the theory.

3.1.4 Yale Shooting Scenario Revisited

Yale shooting scenario which has been used to show certain situation calculus axioms in the previous chapter is given in the Event Calculus in this section to demonstrate this formalism as well. The same domain is chosen to reveal of correspondences.

The initial situation is represented with the predicates *Initially* as follows:

$$Initially(alive(Turkey)) \tag{3-7}$$

$$Initially(\neg loaded(Gun)) \tag{3-8}$$

The effect axioms are defined with the predicates *Initiates* and *Terminates* as given for loading and firing the gun. As it is seen they both define event preconditions[2] and event-fluent pair preconditions.

$$Initiates(load(Gun), loaded(Gun), T) \leftarrow HoldsAt(\neg loaded(Gun), T) \tag{3-9}$$

$$Terminates(fire(Gun), alive(Turkey), T) \leftarrow HoldsAt(loaded(Gun), T) \wedge HoldsAt(alive(Turkey), T) \tag{3-10}$$

$$Terminates(fire(Gun), loaded(Gun), T) \leftarrow HoldsAt(loaded(Gun),T) \tag{3-11}$$

A model which satisfies the theory is given with *Happens* and < (precedes) clauses.

$$Happens(load(Gun), T_1) \tag{3-12}$$

$$Happens(fire(Gun), T_2) \tag{3-13}$$

$$T_1 < T_3 \tag{3-14}$$

$$T_2 < T_3 \tag{3-15}$$

This narrative 3-12 to 3-15 together with the circumscription of predicates *Happens*, *Initiates* and *Terminates*, unique names axioms (fire ≠ load), default axioms (3-1 to 3-6), effect axioms (3-9 to 3-11) and initial state axioms (3-7 and 3-8) entails $HoldsAt(\neg loaded(Gun), T_3)$ and $HoldsAt(\neg alive(Turkey), T_3)$ and this is the only model.

3.2 Compound Events

Events that contain sub-events inside their time frame are compound events. Like GOLOG complex action constructs or HTN methods, they provide

[2] It is possible to specify preconditions in *Happens* clauses as well.

grouping of sub-events with an ordering. Like simple events they can also contain preconditions and recursive definitions.

Consider the following situation. In order to fly from some place to another, the transfer of flight is sometimes required since there is no direct flight to the destination place. However in net effect it should be considered as a whole flight even though it is composed of legs. Assume that a passenger, who is in Ankara, wants to fly to Seattle but there is no direct flight to Seattle. However there are flights from Ankara to Istanbul, Istanbul to Chicago and Chicago to Seattle. This initial setup is represented as follows in the Event Calculus.

$$Initially(at(Ankara)) \tag{3-16}$$

$$flight(Ankara, Istanbul) \tag{3-17}$$

$$flight(Istanbul, Chicago) \tag{3-18}$$

$$flight(Chicago, Seattle) \tag{3-19}$$

Flying from one place to another, the passenger is no longer in his/her previous location but now at destination location of the flight. This is represented with *Fly* event and *at* fluent as follows.

$$Initiates(Fly(O, D), at(D), T) \leftarrow HoldsAt(at(O), T) \tag{3-20}$$

$$Terminates(Fly(O, D), at(O), T) \leftarrow HoldsAt(at(O), T) \wedge O \neq D \tag{3-21}$$

Finally flying from one destination to another is possible if there is a direct flight or there are multiple flights that connect the origin and the destination locations.

$$Happens(Fly(O, D), T_1, T_2) \leftarrow flight(O, D) \tag{3-22}$$

$$\begin{aligned} Happens(Fly(O, D), T_1, T_4) \leftarrow\ & Happens(Fly(O, T), T_1, T_2) \wedge \\ & Happens(Fly(T, D), T_3, T_4) \wedge \\ & T_2 < T_3 \wedge \neg Clipped(T_2, at(T), T_3) \end{aligned} \tag{3-23}$$

Here the *Fly* event is defined to have a precondition and its definition is recursive. Clipped condition is required to avoid interference with other parallel events that would change the location of passenger from the transfer location to another place.

If these axioms are added to the usual axioms of the theory it is valid to have a narrative that concludes the passenger in the final destination via series of flights as follows.

$Happens(Fly(Ankara, Istanbul), T_1, T_2)$ (3-24)

$Happens(Fly(Istanbul, Chicago), T_3, T_4)$ (3-25)

$Happens(Fly(Chicago, Seattle), T_5, T_6)$ (3-26)

$T_2 < T_3$ (3-27)

$T_4 < T_5$ (3-28)

This narrative together with other axioms of the theory entails the result that fluent *at*(*Seattle*) at time point T_6 is true.

3.3 Abduction with Event Calculus

Abduction is a technique for planning used within theorem solvers. It is logically the inverse of deduction and temporal projection. It is used over the Event Calculus axioms to obtain partially ordered sets of events.

In abduction, the goal situation is resolved in reverse direction using the related axioms which might result in proving the goal. Abduction is handled by a second order logical prover which is defined as an abductive theorem prover (ATP) in [32]. It is written in Prolog like GOLOG. This implementation is the basis for this thesis and it is extended to incorporate other necessary features to solve the problem of interest.

3.3.1 Abductive Theorem Prover Implementation

In this implementation axioms are defined inside the predicate *Axiom* in order to gain control of the abduction process. In this respect, it is different from GOLOG implementation in which only the first order logic is used. ATP tries to solve the goal list proving the elements one by one. This technique is very similar to the resolution process of Prolog. Abductive resolution continues until all axioms which are unified with goal clauses are proved.

During the resolution, abducible predicates, which are < (precedes) and *Happens*, are stored in a residue to keep the record of the narrative. This process is depicted in Figure 3-1.

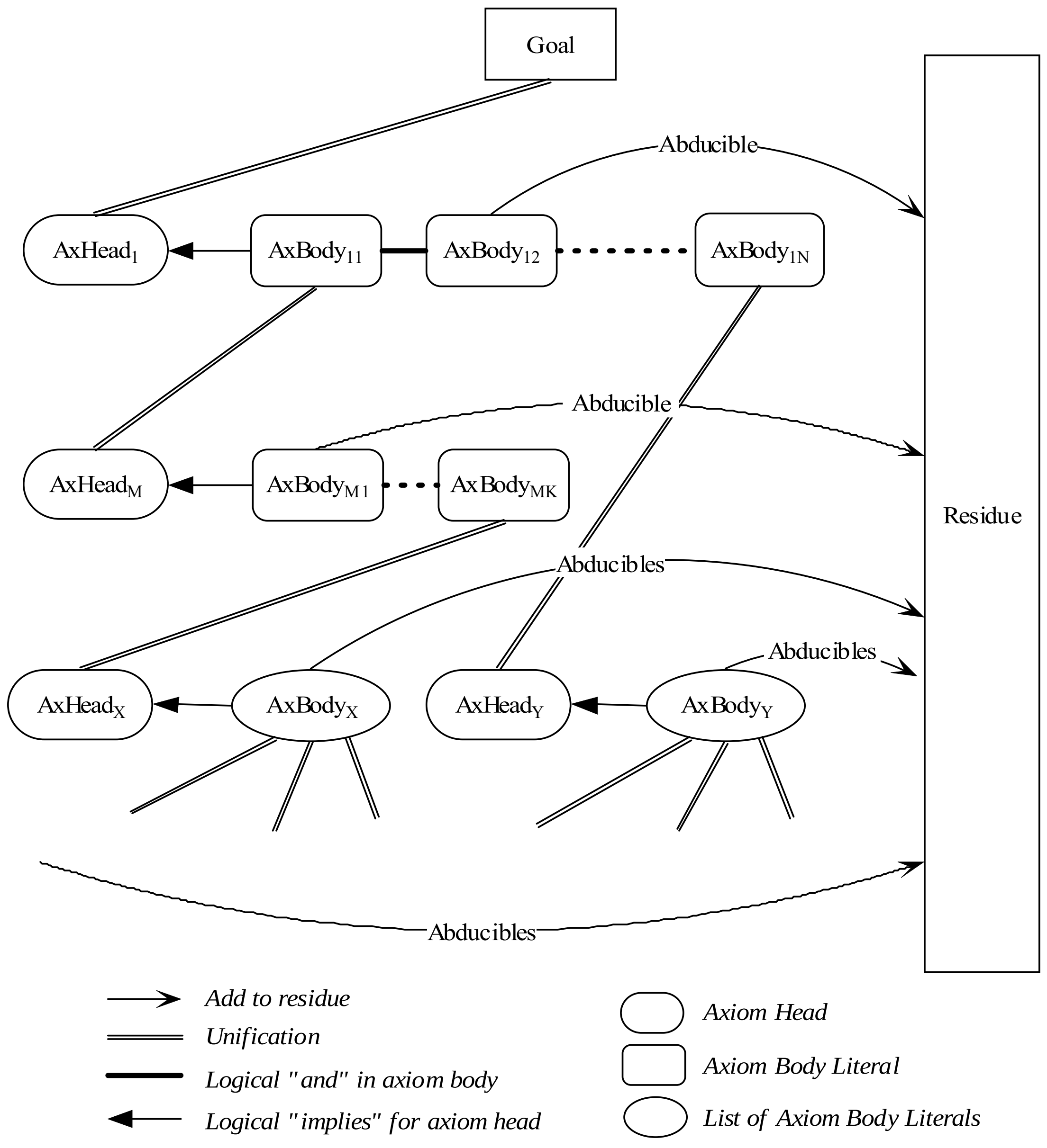

Figure 3-1 Abductive Theorem Proving

The predicate *Ab* is used to denote the theorem prover below. 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 *Ab* solver rule is written. For example assume that an object level axiom is given as follows.

$$AH \leftarrow AB_1 \wedge AB_2 \wedge \ldots \wedge AB_3 \qquad (3\text{-}29)$$

In axiom 3-29, A_H is the head of the axiom and A_{B1} to A_{BN} are the body definition of the axiom. As it is stated before, they are represented with the predicate *Axiom* in the ATP theory and axiom 3-29 is translated to the predicate form in 3-30:

$$Axiom(AH, \{AB_1, AB_2, \ldots, AB_N\}) \qquad (3\text{-}30)$$

Axioms are not directly written as implications and represented inside the predicate *Axiom*. In order to gain the control over resolution process this technique is necessary. During the process axiom bodies are resolved by the *Ab* and this technique allows *Ab* to reach bodies of the axioms. Axiom bodies are resolved by *Ab* but not Prolog itself since *Ab* populates the *abducibles* inside the residue. A simple version of *Ab* solver which solves general axioms like 3-29 is as follows.

$$Ab(GL, RL) \leftarrow GL = \varnothing \qquad (3\text{-}31)$$

$$Ab(\{A\}\ U\ GL, RL) \leftarrow Abducible(A) \wedge Ab(GL, \{A\}\ U\ RL) \qquad (3\text{-}32)$$

$$Ab(\{A\}\ U\ GL, RL) \leftarrow Axiom(A, AL) \wedge Ab(AL\ U\ G, RL) \qquad (3\text{-}33)$$

In this definition *RL* represents the residue list, *GL* denotes goal list, *A* is the axiom head and *AL* is the axiom body. Intuitively, the predicate *Abducible* checks whether the axiom is abducible. If it is so, it is added to the residue[3].

[3] Normally abducible axioms are not subjected to further resolution however in order to support compound events and event preconditions they should be further processed

If it is not an abducible axiom[4] then its body is inserted into the goal list to be resolved with other axioms.

3.3.2 Negated Axioms

Things get complicated when negated axioms are to be proven. They are proven with negation as failure (NAF) technique. In this technique, if the positive of an axiom cannot be proven then the negative of the axiom is said to be proven. However as the residue grows during the resolution, the negative axioms, which are previously proven, might not be proven anymore. This situation occurs since negations of axioms are proven according to the absence of contradicting evidence but the newly added literals might now allow proving the positive of axioms, invalidating the previous negative conclusions. For that reason, whenever the residue is modified, previously proven negated axioms should be rechecked. In order to do this, proven negated axioms have to be recorded in a separate residue like it is done for narratives. The modified version of *Ab* that handles negated axioms is as follows [32].

$$Ab(GL, RL, NL) \leftarrow GL = \varnothing \tag{3-34}$$

$$Ab(\{A\}\ U\ GL, RL, NL) \leftarrow Abducible(A) \wedge Consistent(NL, \{A\}\ U\ RL) \wedge Ab(GL, \{A\}\ U\ RL, NL) \tag{3-35}$$

$$Ab(\{A\}\ U\ GL, RL, NL) \leftarrow Axiom(A, AL) \wedge Ab(AL\ U\ G, RL, NL) \tag{3-36}$$

$$Ab(\{\neg A\}\ U\ GL, RL, NL) \leftarrow Irresolvable(\{A\}, RL) \wedge Ab(AL\ U\ G, RL, \{A\}\ U\ NL) \tag{3-37}$$

In this version, *NL* represents the residue of negated axioms. The predicate *Irresolvable* checks that the negated axiom is not resolvable with the current

[4] Negative test on being abducible is omitted for simplicity.

narrative residue. The predicate *Consistent* checks that none of the negated axioms is resolvable with the current narrative residue using the predicate *Irresolvable* for each negated axiom.

3.3.3 Happened Events

Event definitions might require some other events to have occurred in the past. In order to support such kind of a precondition check another meta axiom is required which checks the current narrative. For this purpose the predicate *Happened* is added to the base implementation of ATP. It checks whether a specific event occurred in a time point prior to the time point supplied. It is defined as in Figure 3-2:

Input Narrative Residue R : [R_T, R_E]

Time Orderings R_T : [$T_A < T_B$, ..., $T_X < T_Y$]

Event Occurrences R_E : [Happens(E_A, T_A), ..., Happens(E_Z, T_Z)]

```
$Happened(E, T, R) \leftarrow Extract(R, R_T, R_E) \wedge$
        $Member(Happens(E, T_0), R_E) \wedge$
        $Before(T_0, T, R_T)$
```

where

```
$Before(T_0, T_1, R_T) \leftarrow Member(T_0 < T_1, R_T) \vee$
        $(Member(T_0 < T_X, R_T) \wedge Before(T_X, T_1, R_T))$
```

Figure 3-2 Happened Predicate

It takes the residue as an argument to check if this event has occurred at a time point T_0 prior to the time supplied. In the formulation above, R_T denotes the time orderings of the events which are stored in R_E. Both R_T and R_E are internal sub-lists of the residue *R* and the predicate *Extract* is used to extract these sub-lists. This meta-predicate is useful for decisions where complete state space is not known and forward decomposition on event definitions are employed to reach the final state.

However it might hinder the execution in backward planning where the final state known and narrative yielding this state is required. When *Happened* is queried during the reverse resolution, residue might not contain the required prior events simply because abduction process has not reached to that time point yet. For that reason, resolutions on the predicate *Happened* fail while an answer can be found if forward planning is applied. Thus this predicate violates the completeness of the ATP since it fails to find the existing answer for these problems during backward planning. Yet despite these consequences it might be very useful for forward planning since it brings certain control over the sequencing of events. For instance, synchronization constructs which are generally used inside workflows can easily be modeled inside the Event Calculus with the help of this predicate.

As an example the following rule states that a conference reservation can only be made if the flight reservation and hotel reservation have been done previously. The residue arguments are omitted for simplicity.

$$\begin{aligned}&\mathit{Happens}(\mathit{BookConference}, T) \leftarrow \\ &\qquad \mathit{Happened}(\mathit{BookFlight}, T) \wedge \mathit{Happened}(\mathit{BookHotel}) \qquad (3\text{-}38)\end{aligned}$$

3.3.4 Goal Check

Another improvement on the ATP is not solving the fluents which are already implied by the current residue. In order to employ this technique whenever a fluent to be satisfied is encountered by the resolution process of the ATP, the current residue is checked. If the current residue already contains the goal at hand then no further action is taken.

This technique is useful for reducing the resolution space hence reducing the number of steps in the resulting plan. However this technique also suffers from the same problem of proving the negated goals. For instance, if a fluent is found to be initiated at a time point prior to the requested time point then it is assumed that that fluent holds at the requested time point and no further resolution is performed. However, further additions to the residue prior to the requested time point and later than the initiation time point might have terminated the fluent. Hence, the previous assumption fails and the resolution continues with an invalid assumption.

In order to avoid this problem the residue of negated axioms is modified to contain the fluents that have been declipped[5] before the requested time point. Since the negated axiom residue is checked whenever the narrative residue is modified, it is impossible to add inconsistent entries to the residues. Thus the resolution process is kept sound. A vanilla representation of this situation is given below:

$$\begin{aligned}
&Ab(\{HoldsAt(F, T)\} \cup GL, RL, NL) \leftarrow \\
&\quad Member(Clipped(T_1, F, T_2), NL) \wedge \\
&\quad T_2 < T \wedge T_2 < T_3 \wedge T_2 < T_4 \wedge \\
&\quad \neg Member(Declipped(T_1, F, T_2), NL) \wedge \\
&\quad Remove(Clipped(T_1, F, T_2), NL, NLR) \wedge \\
&\quad Ab(GL, RL, NLR \cup \{Clipped(T_1, F, T)\}) \qquad (3\text{-}39)
\end{aligned}$$

[5] In fact the predicate clipped is used since it is inserted in the residue of negated axioms.

Here the predicate *Remove* removes the element from the list and unifies the last argument with the reduced list. The predicates *Clipped* and *Declipped* are used to hold the fluent valuations in the residue of negated axioms.

CHAPTER IV

WEB SERVICE COMPOSITION WITH THE EVENT CALCULUS

4.1 Composition with Abductive Planning

The Event Calculus can be used for planning as it is theoretically explained in the CHAPTER III. The ordering of events which constitutes a valid flow of execution is obtained by abduction. This ordering is obtained with the resolution of axioms that defines the theory. Once the axioms are formed abducible literals can be collected during the resolution process.

4.1.1 Web Services as Events

In order to incorporate Web Services to the Event Calculus, the first thing to do is to bind the external services to the theory of the problem domain. This integration is accomplished by modeling Web Services as events with parameters. The external Web Service calls are modeled as predicates and parameters of the predicates are organized as input, output and if desired, as the result of the Web Service. For instance, assume a Web Service operation returns the availability of a flight between two locations. Its corresponding event is given in Figure 4-1.

```
Operation GetFlights
      Inputs(Origin - City,
             Destination - City,
             FlightDate - Date)

      Outputs(FlightNumbers – List of Integers)

Happens(GetFlights(O, D, FD, FNL), T1, T1) ←
      Ex_GetFlights(O, D, FD, FNL)
```

Figure 4-1 Web Service to Event Translation

As it is seen, the Web Service operation *GetFlights* is translated to the event *GetFlights*. The inputs and outputs of the Web Service are translated as parameters of the event. The parameters are populated with help of the predicate *Ex_GetFlights*. This predicate is used as a precondition for the event and for that reason it is invoked anytime it is added to the plan. In order to resolve literals which are non-axiomatic assertions such as conditions or external calls *Ab* is extended to contain the following rule:

$$Ab(\{L\} \cup GL, R, N) \leftarrow \neg Axiom(L) \wedge L \wedge Ab(GL, R, N) \qquad (4\text{-}1)$$

In this rule *L*, *GL*, *R* and N denote the non-axiomatic literal, the goal list, the narrative residue and the negation residue respectively. If a non-axiomatic literal is encountered then *Ab* directly tries to prove the literal and if it is successful it continues with rest of the goal list.

In ATP implementation, the external call bindings like the predicate *Ex_GetFlights* are loaded from an external module that is written in C++ programming language. After invoking the associated service, flight numbers are unified with *FNL*, the last parameter of *GetFlights* event, which is an atom list.

In the scope of this thesis, the data transformations are not considered and intrinsic types are assumed to be used for both, programming languages. However it is possible to extend the data types using an object-oriented Prolog. Also automatic binding to external web services are considered to be out of the scope of the thesis.

4.1.2 Plan Generation

The implementation of abductive Event Calculus, ATP, returns a valid sequence of event timings that leads to the resulting goal, but if there is more than one solution they are obtained with the help of the backtracking of Prolog. In that situation it could be thought that there is more than one solution that leads to the goal but since they are different solution of the same problem, they can be thought as different braches of a more general solution or they are the different braches of the same plan. For instance, assume that the following event sequence is generated after successful resolution process.

$$Happens(E_1, T_1) \quad (4\text{-}2)$$

$$Happens(E_2, T_2) \quad (4\text{-}3)$$

$$Happens(E_3, T_3) \quad (4\text{-}4)$$

$$T_1 < T_2 < T_3 \quad (4\text{-}5)$$

It is concluded that when executed consecutively, the ordered set $\{E_1, E_2, E_3\}$ generates the desired effect to reach the goal. But as it is explained in Section 4.1.1, the axioms of the events may contain non-axiomatic literals in the body and if they are Web Service calls then the desired effect might not be generated as the plan is executed. For that reason, if an event sequence fails to execute the alternative solutions should be examined. In order to do such a maneuver, the executer should have a tree like plan where proceeding with alternative paths is possible. Assume that the following other totally ordered sequences of events yield the same result with narrative given in 4-2 to 4-5:

$$\{E_1, E_5, E_4\} \tag{4-6}$$

$$\{E_1, E_2, E_4\} \tag{4-7}$$

$$\{E_6, E_7\} \tag{4-8}$$

When these solutions are combined, a workflow which describes a Web Service composition is formed as depicted in Figure 4-2.

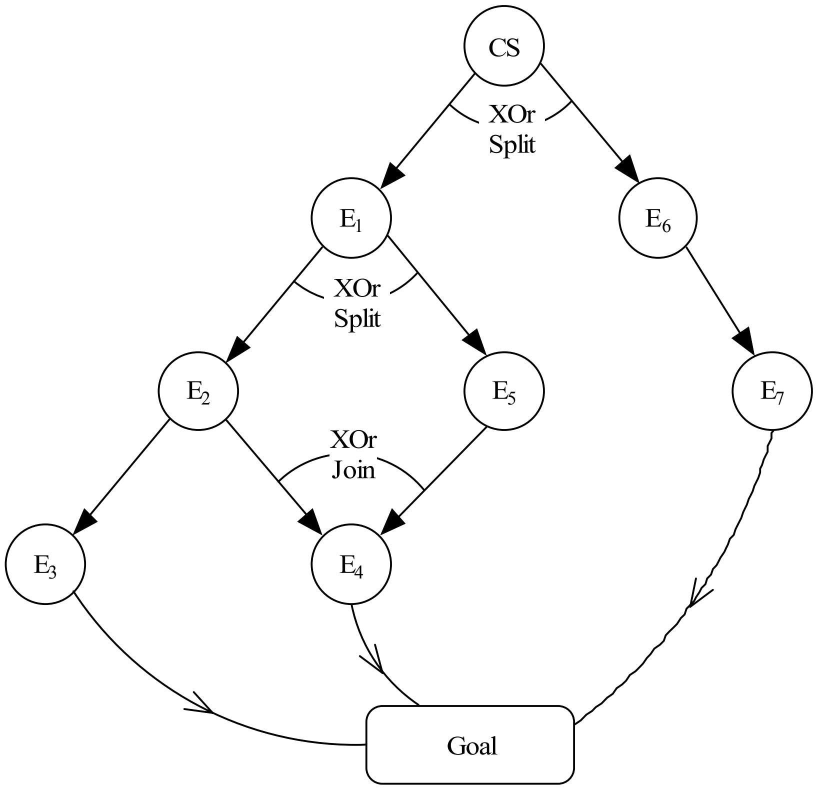

Figure 4-2 Multi-branch Composition

In this figure events are the nodes of the workflow and *CS* is representing the start of composition. There are several possible alternatives at certain points. These are represented in figure as *Exclusive-Or-Splits* (*XOr*) since any of the choice is possible but only one of the braches is taken. Also several alternative paths are joined in the graph since their last event is the same and these are depicted as *XOr-Joins*. *XOr-Joins* mandate that only one of the branches is active at the joining side. This workflow is a composition plan for the problem it solves.

4.1.3 Concurrency Of Events

ATP generates narratives for the solution of the problem domain which are partially ordered set of events. The partial ordering is arisen since events are relativized with respect to time instead of each other. For that reason, during the resolution, separate and unrelated events are thought to be concurrent since there is no precedes relation between them. Assume that ATP has generated a narrative as given below:

$$Happens(E_1, T_1) \tag{4-9}$$

$$Happens(E_2, T_2) \tag{4-10}$$

$$Happens(E_3, T_3) \tag{4-11}$$

$$Happens(E_4, T_4) \tag{4-12}$$

$$T_1 < T_2 < T_4 \tag{4-13}$$

$$T_1 < T_3 < T_4 \tag{4-14}$$

Since there is no resolved precedence relation between E_2 and E_3 they are assumed to be concurrent while both of the two orderings $T_2 < T_3$ and $T_3 < T_2$ generate the same desired effect. If this is the only ordering generated by the ATP then as a plan it can be diagramed as in Figure 4-3.

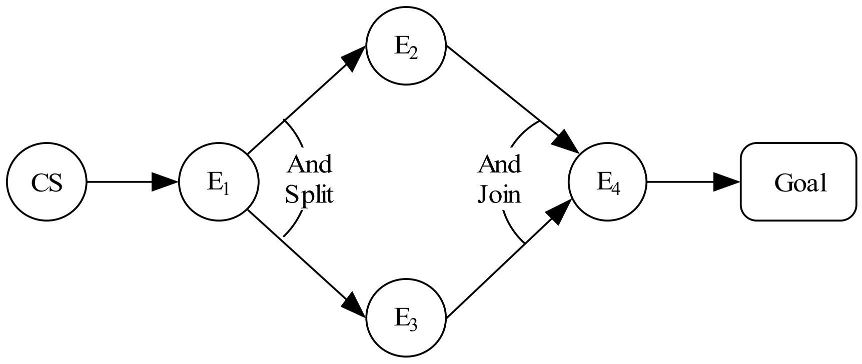

Figure 4-3 Concurrent Composition

In this workflow, concurrent events are depicted as *And-Split* since both of the branches should be taken after the event E_1. Before the event E_4 *And-Join* is required since both of E_2 and E_3 should be executed.

This kind of workflow modeling has been formalized by using the Event Calculus and Action Description Languages in previous studies [15, 4] for the specification and execution of workflows. In this thesis however they are describing the plan that has been generated by ATP.

4.2 PDDL to Event Calculus Translation

Being a widely accepted specification for planning domains PDDL is chosen for translation to express the planning capabilities of the Event Calculus. This approach is also used for Estimated Regression Planners as it is described in [21] and also in the Section 2.4.4.1.

As it is argued in [21], simplifying and restricting the input language allows the planner to deal with a defined domain. Besides, there are translators from DAML-S to PDDL meaning that once planning is possible in

PDDL domain it is also possible to plan the Web Service Compositions in the planning domain.

In this section, the preliminary translation algorithm is given. Only core of the PDDL is translated to the Event Calculus. Full translation is not possible since PDDL is not solely a logical language, yet, to the best of our knowledge there is no planner at all which fully supports all *requirements* of the PDDL.

4.2.1 Requirements

This translation is scoped over several *requirements* of PDDL. These are *:strips*, *:equality*, *:conditional-effects, :numbers, :fluents, :domain-axioms, :open-world, :true-negation* and *:existential-preconditions*.

The most primitive domain is *:strips* which is the core of the PDDL. It implies support for actions. In our case, actions will be represented with the events. Support for the requirements *:equality* and :*numbers* are satisfied with the built-in support of Prolog for equality operator and arithmetical operations. The requirement of *:conditional-effects* is supported with the the effect axioms of the Event Calculus whose preconditions can be defined in the body of axioms.

The requirements of *:fluents, open-world* and *:true-negation* are also supported since fluents with negative valuations can be used in the Event Calculus.

4.2.2 Types

Types of the languages are defined as predicates of first order objects in the Event Calculus. In PDDL, type hierarchies can be defined as it is in object-oriented languages. These type hierarchies can be defined in the Event Calculus as Prolog implications. For instance, assume that there are two types: *shape* and *rectangle* where *rectangle* is derived from *shape*. *Move* event is possible for first order objects that are *Shape*. Since each *rectangle* is a *shape*, each rectangle is movable. This situation is depicted in the Event Calculus as follows.

$$Shape(X) \leftarrow Rectangle(X) \quad (4\text{-}15)$$

$$Happens(Move(X), T) \leftarrow Shape(X) \quad (4\text{-}16)$$

More complicated types are also present in PDDL under *:type* requirement but our translation does not support all of them.

4.2.3 Axioms

Axioms of the PDDL can be translated as axioms of the Event Calculus almost without any difficulty. In order to define the axioms of PDDL, state constraints of the Event Calculus are used. State constraints are axioms of the Event Calculus which manipulate valuation of fluents when certain conditions are met. The most apparent difference with the effect axioms is that effect axioms alter the valuation of fluents when a triggering event fires; however state constraints alter the valuations of fluents when a sought condition or situation (state) occurs. An example state constraint is as follows.

$$HoldsAt(available(Robot), T) \leftarrow HoldsAt(free(Gripper)) \wedge Has(Robot, Gripper) \quad (4\text{-}17)$$

This state constraint says that all robots which have at least one free gripper are available (for carrying objects). As it is seen the fluent *available* is not related to an event[6] but to a state (in fact, a subset of the problem domain state space).

[6] Since dependent fluents change their value with events, state constraint fluents are, in fact, indirectly related with events.

A PDDL axiom is very similar to the state constraint of the Event Calculus. Its structure and two examples [36] are given in Figure 4-4.

```
<axiom-def> ::= (:axiom
                  :vars (<typed list (variable)>)
                  :context <goal proposition>
                  :implies <literal(term)>)

(:axiom
      :vars (?x ?y - physob)
      :context (on ?x ?y)
      :implies (above ?x ?y))

(:axiom
      :vars (?x ?y - physob)
      :context (exists (?z - physob)
              (and (on ?x ?z) (above ?z ?y)))
      :implies (above ?x ?y))
```

Figure 4-4 PDDL Axioms

PDDL axioms have three parts: *vars* which lists the objects used inside the axiom, *context* which is the *if-condition* of the axiom and *implies* which is the *then-statement* of the axiom. This axiom is translated to the Event Calculus state constraint as given in (4-18).

$$HoldsAt(<literal(term)>) \leftarrow <goal\ proposition> \quad (4\text{-}18)$$

Literals can be fluents but they must not have effect axioms which might violate the execution of ATP. The problem arises from the fact that, fluents seem to have their valuation changed with these axioms without actually initiation or termination of fluents in negated axioms residue. The translations of PDDL axiom examples are given in Figure 4-5

```
HoldsAt(above(X, Y), T) ← PhysOb(X) ∧ PhysOb(Y) ∧
      HoldsAt(on(X, Y), T)

HoldsAt(above(X, Y), T) ← PhysOb(X) ∧ PhysOb(Y) ∧ PhysOb(Z) ∧
      HoldsAt(On(X, Z), T) ∧ HoldsAt(above(Z, Y), T)
```

Figure 4-5 PDDL Axiom Translation

For the clause *exists* appearing in the PDDL axiom there is nothing to do since Prolog handles the condition itself by finding some object which satisfies the condition or failing otherwise.

4.2.4 Actions

The actions of PDDL are represented with the effect axioms in the Event Calculus. Actions have parameters, preconditions and effects just like events of the Event Calculus. Its structure and an example [36] are given in Figure 4-6.

```
<action-def> ::= (:action (<name>)
                 :parameters (<typed list (variable)>)
                 :vars (<typed list (variable)>)
                 :precondition <goal proposition>
                 :effect <effect proposition>

(:action move
      :parameters (?briefcase – physobj ?m ?l – location)
      :precondition (and (at B ?m) (not (= ?m ?l)))
      :effect (and (at ?briefcase ?l) (not (at ?briefcase ?m))
                 (forall (?z)
                       (when (and (in ?z)
                                  (not (= ?z briefcase)))
                             (and (at ?z ?l) (not (at ?z ?m))))
```

Figure 4-6 PDDL Actions

The given example is chosen from a common problem which is the briefcase world. In this problem, moving the briefcase indirectly moves the items inside the briefcase.

In order to translate PDDL action to an Event Calculus effect axiom, preconditions are used as preconditions of the effect axiom. For each literal in the conjunction of the effect clauses, one effect axiom is written. The condition appearing in the PDDL action is added to the precondition of the corresponding effect axiom. Finally for *forall* clauses nothing is necessary since the axiom is applicable for all objects satisfying the conditions. According to these conversion rules the translated action definition in the Event Calculus is given in Figure 4-7.

```
Happens(move(B, M, L), T) ← PhysOb(B) ∧ Location(M) ∧
      Location(L)

Initiates(move(B, M, L), at(B, L), T) ←   HoldsAt(at(B, M), T) ∧
      M ≠ L

Initiates(move(B, M, L), ¬at(B, M), T) ← HoldsAt(at(B, M), T) ∧
      M ≠ L

Initiates(move(B, M, L), at(Z, L), T) ← HoldsAt(at(B, M), T) ∧
      M ≠ L ∧ HoldsAt(in(Z), T) ∧ Z ≠ B

Initiates(move(B, M, L), ¬at(Z, M), T) ← HoldsAt(at(B, M), T) ∧
      M ≠ L ∧ HoldsAt(in(Z), T) ∧ Z ≠ B
```

Figure 4-7 PDDL Action Translation

Four effect axioms are written in the Event Calculus for four literals in the conjunction of *move* action effect. For the items inside the briefcase, *when* condition is appended to the preconditions of related effect axioms in the translation algorithm.

4.2.5 Problems

PDDL problems are composed of some initial state statements and a goal statement as it is explained in Section 2.2.3. The initial state is translated to the Event Calculus with the help of the axioms *Initially*. Finally, the goal is translated to the Event Calculus like other *goal descriptors* as a conjunction of literals and it is given to ATP for solving. The Event Calculus representation of the PDDL problem provided in Section 2.2.3 is given in Figure 4-8 as an example translation.

```
room(RoomA)
room(RoomB)
ball(Ball1)
ball(Ball2)
gripper(Left)
gripper(Right)

Initially(at-robby(RoomA))
Initially(at(Ball1, RoomA))
Initially(at(Ball2, RoomA))
Initially(free(Left))
Initially(free(Right))

←HoldsAt(at(Ball1, RoomB), Tf) ∧ HoldsAt(at(Ball2, RoomB), Tf)
```

Figure 4-8 PDDL Problem Translation

The first group of axioms that is given in Figure 4-8 defines the types of objects. The second group of axioms defines initial situation. Finally the goal is given as a rule without a head. T_f in the goal statement denotes the time of the final situation where there is no other time point preceded by it.

An example translation of the PDDL domain and the problem given in the Section 2.2 is given in APPENDIX C. It is a Prolog program for the ATP interpretation.

4.3 Quality Of Service Composition

Another approach for the Web Service Composition is building the composition based on Quality of Service (QoS) parameters [40, 26]. In this technique, QoS parameters are used as the composition criteria. Web Services are selected from those who conform to the QoS parameters.

For instance, assume a Personal Computer (PC) configuration tool which provides selection of QoS parameters for the peripherals that constitutes PC. Customers specify preferences for the various parameters for the peripherals and based on these preferences the PC is built. Customers also specify the constraints for maximum price and latest shipment date for the computer.

Considering the customer preferences and a predefined flow for peripheral integration, the PC assembler tries to build a computer that obeys the customer constraints. Automating the assembly process provides the following advantages:

- Customer sees whether the specified QoS parameters constitute a PC.
- If it is possible to build the PC, seller gets a peripheral integration plan.
- Peripherals are ordered from the retailers automatically and seller spends only technical effort and focuses only on assembling the PC.

In order to do this Web Service Composition, retailers should publish QoS parameters, stock availability and shipment date for the sold items via Web Services. Furthermore a template integration scheme which describes the abstract assembly order should be known. Based on this abstract integration flow a concrete build plan is generated. Finally, the compatibility requirements for the peripherals should also be checked during the selection of them. These requirements can also be captured as the compatibility of QoS parameters and they can be added to the abstract flow.

For instance, if a motherboard with a X86 architecture is selected then a central processing unit (CPU) should be of X86 architecture as well. In order to satisfy the requirement whenever CPU is selected its compatibility with motherboard should be checked. This check is performed with the help of QoS parameters which specify the suited architecture.

4.3.1 Event Calculus Planning for the QoS Composition

This example is converted to an Event Calculus planning problem keeping the perhipheral compatibility issues minimum. First a generic flow which outlines the integration order is defined. A generic flow is given in Figure 4-9.

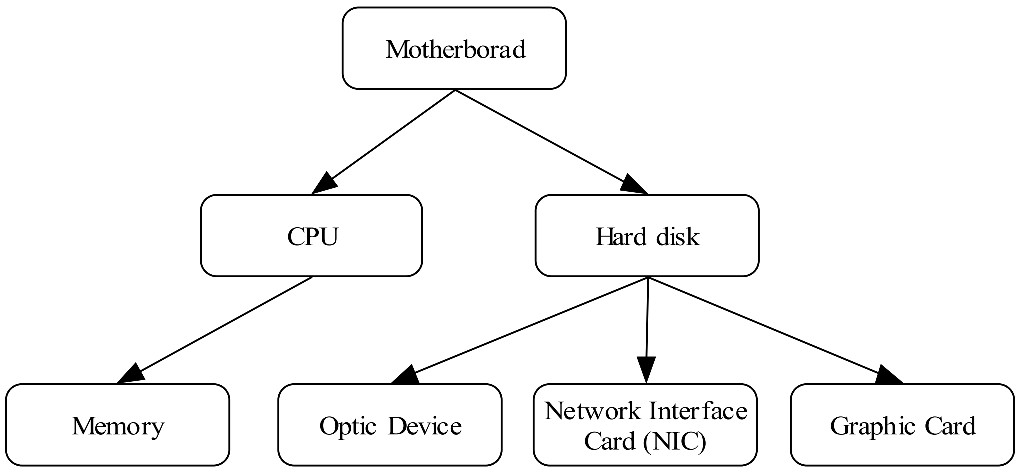

Figure 4-9 PC Configuration Integration Order

In this flow the peripherals that are going to be integrated sequentially are represented with nodes connected with successive arrows. Parallel arrows define the possible concurrent integrations. It is possible to define other integration orders as well.

Table 4-1 summarizes the QoS parameters used for the configuration problem demonstrated in this section.

Table 4-1 Peripherals QoS Parameters

Peripheral	QoS Parameters
Motherboard	Minimum Memory Speed (MHz), CPU Type (AMD/Intel), Harddisk Link (SCSI/IDE), Minimum Graphic Card Speed (X), Price
Harddisk	Link Type (SCSI/IDE), Capacity (Gigabyte), Price
Memory	Speed (Mhz), Capacity (Megabyte), Price
CPU	Speed (GHz), CPU Type (AMD/Intel), Price
Optic Reader	Type (CD/DVD/CD-RW/DVD-RW), Price
NIC	Max Speed (Mbits/s), Price

The goal is to configure a PC which suits the user constraints. PC is configured whenever all the peripherals are configured and PC-wide constraints like build date and total price are checked. The Web Service calls with specified QoS parameters are modeled as events as usual. These events trigger external calls which wrap Web Service access and unify the QoS parameters. Whenever an item is configured a fluent is initiated which carries the configured item QoS parameters. This is one way to share the information generated by the events or situations. A part of the effect axioms which initiate these fluents are given in Figure 4-10.

```
Motherboard quotation
Initiates(EvQuoteMB,
          FlMBQuoted(MemSpd, CpuTyp, HdTyp, GcSpd, MbPrc, MbDate),
          T) ←
      HoldsAt(FlPCUnconfigured, T) ∧
      ExQuoteMB(MemSpd, CpuTyp, HdTyp, GcSpd, MbPrc, MbDate) ∧
      MBUserConstr(MemSpd, CpuTyp, HdTyp, GcSpd, MbPrc, MbDate)

CPU quatation
Initiates(EvQuoteCPU,
          FlCPUQuoted(CpuSpd, CpuTyp, CpuPrc, CpuDate),
          T) ←
      HoldsAt(FlMBQuoted(_, CpuTyp, _, _, _, _) , T) ∧
      ExQuoteCPU(CpuSpd, CpuTyp, CpuPrc, CpuDate) ∧
      CPUUserConstr(CpuSpd, CpuTyp, CpuPrc, CpuDate)

Hardisk quatation
Initiates(EvQuoteHD,
          FlHDQuoted(HdTyp, HdSz, HdPrc, HdDate),
          T) ←
      HoldsAt(Fl_mbQuoted(_, _, HdTyp, _, _, _) , T) ∧
      ExQuoteHD(HdTyp, HdSz, HdPrc, HdDate) ∧
      HDUserConstr(HdTyp, HdSz, HdPrc, HdDate)

Other peripherals...

PC Configuration
HoldsAt(FlPCConfig, T) ←
      HoldsAt(FlMBQuoted(MemSpd, CpuTyp, HdTyp, GcSpd, MbPrc,
                         MbDate), T) ∧
      HoldsAt(FlCPUQuoted(CpuSpd, CpuTyp, CpuPrc, CpuDate), T) ∧
      HoldsAt(FlMemQuoted(MemSpd, MemSz, MemPrc, MemDate), T) ∧
      HoldsAt(FlHDQuoted(HdTyp, HdSz, HdPrc, HdDate), T) ∧
      HoldsAt(FlORWQuoted(OrwTyp, OrwPrc, OrwDate), T) ∧
      HoldsAt(FlGCQuoted(GcSpd, GcSz, GcPrc, GcDate), T) ∧
      HoldsAt(FlNICQuoted(NicSpd, NicPrc, NicDate), T) ∧
      PcPrc = MbPrc + CpuPrc + MemPrc + HdPrc +
              OrwPrc + GcPrc + NicPrc ∧
      PcDate = Maximum(MbDate, CpuDate, HdDate,
                       OrwDate, GcDate, NicDate) ∧
      PCUserConstr(PcPrc, PcDate)
```

Figure 4-10 PC Configuration with the Event Calculus

In Figure 4-10, events are prefixed with “Ev”, fluents are prefixed with “Fl” and external Web Service calls are prefixed with “Ex”. Also, the parameters Represented with an underscore “_” are *don’t care* ones like in Prolog. In the beginning *FlPCUnconfigured* fluent is true which the starting point of the configuration is. With the help of subsequent effect axioms, PC is configured whenever *FlPCConfig* fluent is initiated. In other words, it is the goal fluent to be searched. Only CPU, motherboard and hard disk effect axioms are given since the rest is similar to them.

Whenever the user specifies constraints they are complied and added to the theory as axioms (*MBUserConstr, CPUUserConstr, HDUserConstr, ..., PCUserConstr*). Their heads are added to the effect axioms of relevant peripherals as preconditions. Furthermore compatibility constraints are specified by using the same variable in rule bodies. For example, in CPU quotation, CPU type is checked to be the same with the motherboard CPU type in the predicate *HoldsAt*. Sample user constraint axioms are given in Figure 4-11.

```
GCUserConstr(GcSpd, GcSz, GcPrc, GcDate) ←
            GcSpd >= 4 ∧ GcSz >= 128

NICUserConstr(NicSpd, NicPrc, NicDate) ←
            NicSpd >= 100

PCUserConstr(PcPrc, PcDate) ←
            PcPrc < 1000 ∧ PcDate < 10
```

Figure 4-11 User Constrains

In this example, the user wants a PC whose price is below 1000 (YTL) and whose shipment can be completed in 10 days. Also graphic card bus speed should be at least 4X and its memory size should be at least 128 megabytes. Finally NIC should operate in at least 100Mbits/s. The rest of the constraints are added without any body and for that reason they do not have any effect. When the user constraints are added to generic composition, a theory which is ready for planning is obtained.

CHAPTER V

WEB SERVICE EXECUTION WITH THE EVENT CALCULUS

5.1 OWL-S to Event Calculus Translation

One of the most promising leaps on automating the Web Service Composition is taken with the formation of OWL-S language. In this language, Web Services are abstracted, composed and bound to concrete providers. The abstraction helps the service seekers to easily understand the supplied features of available services. Ability to express composition allows the specification of the interaction scenarios of the Web Services. OWL-S also allows concrete realization of Web Service interfaces allowing clients physically connect and communicate with the advertised services.

5.1.1 Composite Processes

Web Services are composed of a series of operations, which atomically provide certain functions. Service interactions can be as simple as a single operation invocation. For instance, an organization, *RandomDotOrg* (http://www.random.org), generates and returns random numbers with a single Web Service operation. However they can be as complicated as a multi-department electronic commerce site for shopping, where catalog browsing, item selection, shipment arrangements and payment selection are accomplished by invoking a series of operations. *Amazon Web Service* (http://www.amazon.com) provides such features with a large set of Web Service operations. It is also one of the first industry corporations that have adopted OWL-S language together with WSDL.

When the number of operation invocations required for a certain service increases the interaction scenarios should be known by the client. For the automation purposes these scenarios are standardized in the *Service Model* section of OWL-S. An example *Service Model* is given in APPENDIX B.

In this section, ordinary Web Service operations are modeled as *Atomic Processes*. Several atomic processes constitute a *Composite Process w*hen connected with the flow control constructs of OWL-S. A *Composite Process* is an encapsulation for the world altering effects which are generated by the execution of the flow it describes. However although they encapsulate the complexity beneath, they are not executable as single-step like the atomic processes because there is no corresponding single Web Service operation. They just provide a structural ordering and logical grouping in order to be used in other composite processes for certain function or purpose and they describe the resulting effects.

5.1.2 Translation of Composite Processes

If an automated system requires the provided service it should execute the composite processes as they are defined in the OWL-S, supplying the intermediate inputs to the atomic services nested under them. This section proposes the use of the Event Calculus as an executer for the composite processes.

With the help of compound events in the Event Calculus [32], composite processes can be modeled. Like the composite processes, compound events provide the grouping of sub-events. After they are translated to the corresponding axioms they can be executed with the Event Calculus as a middle ground executer.

Composite processes are composed of control constructs which closely resembles to the standard workflow constructs. Since further composite processes can be used inside a composite process, the translation is recursively applied until all composite processes are replaced with the corresponding axioms that contain atomic processes.

5.1.2.1 Atomic Processes

Atomic processes are translated into simple events of the Event Calculus. The inputs and outputs are converted to the parameters of the events. The preconditions are defined in the effect axiom bodies. The outputs and effects are generated with the effect axiom heads. The conditional effects are generated with conditions being added to the preconditions. Representation of an atomic process of OWL-S is illustrated in Figure 5-1.

```
Atomic Process<$A$, $V$, $P$, $E$, $O$, $E_C$, $O_C$>
      $A$ : Atomic Process Functor

      $V$ : Set of Inputs
            $\{V_1, V_2, ..., V_N\}$

      $P$ : Preconditions
            Conjunction of Literals $(P_1 \wedge P_2 \wedge ... \wedge P_M)$

      $E$ : Effects
            Conjunction of Literals $(E_1 \wedge E_2 \wedge ... \wedge E_K)$

      $O$ : Outputs
            Set of Outputs $\{O_1, O_2, ..., O_L\}$

      $E^C$ : Conditional Effects
            Set of literals $\{E^C_1, E^C_2, ..., E^C_R\}$
            where each $E_{Ci}$ has a condition such as
            $E_{Ci} \leftarrow BE_{Ci}$ : $BE_{Ci}$ are conjunction of literals

      $O_C$ : Conditional outputs
```

Figure 5-1 Atomic Process Definition of OWL-S

This definition is converted to the Event Calculus axioms in a similar way that the actions of the PDDL are translated. An event with the same name of the atomic process *A* is created and the effect axioms are defined according to the preconditions and effects. The translation is given in Figure 5-2.

$$
\begin{aligned}
&\mathit{Initiates}(A(V, O), E_i, T) \leftarrow \mathit{HoldsAllAt}(P, T) \wedge \\
&\qquad \mathit{Invoke}(A, V, O, T) \\
&\quad \text{where } E_i \in E^+ \text{ (positive literals of } E\text{)} \\[1em]
&\mathit{Terminates}(A(V, O), E_i, T) \leftarrow \mathit{HoldsAllAt}(P, T) \wedge \\
&\qquad \mathit{Invoke}(A, V, O, T) \\
&\quad \text{where } E_i \in E^- \\[1em]
&\mathit{Initiates}(A(V, O), E^C{}_i, T) \leftarrow \mathit{HoldsAllAt}(P, T) \wedge \\
&\qquad \mathit{HoldsAllAt}(BE^C{}_i, T) \wedge \mathit{Invoke}(A, V, O, T) \\
&\quad \text{where } E^C{}_i \in E^{C+} \\[1em]
&\mathit{Terminates}(A(V, O), E^C{}_i, T) \leftarrow \mathit{HoldsAllAt}(P, T) \wedge \\
&\qquad \mathit{HoldsAllAt}(BE^C{}_i, T) \wedge \mathit{Invoke}(A, V, O, T) \\
&\quad \text{where } E^C{}_i \in E^{C-} \\[1em]
&\text{where} \\
&\mathit{HoldsAllAt}(\{F_1, F_2, ..., F_Z\}, T) \leftrightarrow \mathit{HoldsAt}(F_1, T) \wedge \\
&\qquad \mathit{HoldsAt}(F_2, T) \wedge ... \wedge \mathit{HoldsAt}(F_Z, T)
\end{aligned}
$$

Figure 5-2 Atomic Process Translation

As it is given in Figure 5-2, the meta predicate *HoldsAllAt* has an equivalent effect of conjunction of *HoldsAt* for each fluent that *HoldsAllAt* covers.

The predicate *Invoke* is used in the body of effect axioms to generate the desired outputs. It takes the name of the atomic process, input parameters and unifies the outputs with the results of the corresponding Web Service operation invocation. The final parameter, time tag, is used to cache results for operations. If operations are not cached according to their time points then different effect axioms of the same atomic process would be associated with different outputs which might result conflicting effects when the World Altering Web Service operations are invoked to collect outputs.

The atomic processes can have conditional outputs as well as conditional effects, however the Event Calculus cannot handle the conditional outputs and therefore its translation is not given. Our assumption is that the translation process does not involve atomic processes with conditional outputs.

5.1.2.2 Composite Process Translation

Composite Processes combine a set of processes (either atomic or composite) with a control construct. More than one construct can be used in nested structures. A composition which is composed of nested structures is given in Figure 5-3.

In this example a composite process with a sequence of sub-composite processes are given. Two of the sub-processes are atomic processes and the others are composite processes. Split, Join and Repeat-While control constructs are used in this composite process.

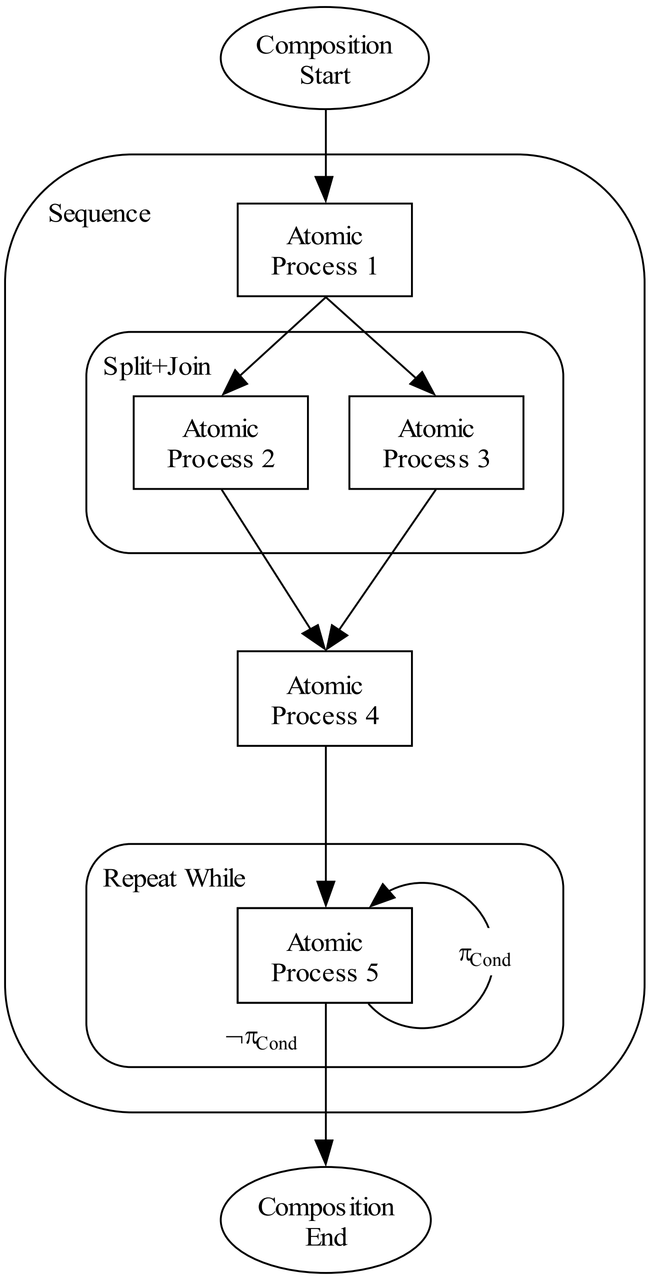

Figure 5-3 Composite Process Example

In this section all such composite and atomic constructs are translated into the Event Calculus axioms. First the definition of the construct is given then its translation will be presented. Constructs are originally defined in XML (to be more precise in RDF) document structure however since they are not much readable and space consuming their abstract equivalents will be given in the subsequent sections. An example of the *Sequence* construct is given in APPENDIX B.

Sequence

The *Sequence* construct contains the set of all component processes to be executed in the order. The definition of the composite process containing a *Sequence* control construct is given in Figure 5-4.

```
Sequence Composite Process<C, V, P, S>
      C : Composite Process Functor

      V : Set of Inputs
             {$V_1$, $V_2$, ..., $V_N$}

      P : Preconditions
            Conjunction of Literals ($P_1 \wedge P_2 \wedge ... \wedge P_M$)

      S : Sequence of Sub-Processes
            Ordered set of processes {$S_1$, $S_2$, ..., $S_K$}
```

Figure 5-4 Sequence Composite Process

The translation is accomplished through the use of compound events in the Event Calculus which contains sub-events. The sequence of events are triggered from the body of the compound event and the ordering between them is ensured with the predicate < (precedes). The formulation is given in Figure 5-5.

$$\begin{aligned}
&Happens(C, T_1, T_N) \leftarrow HoldsAllAt(P, T_1) \wedge \\
&\quad Happens(S_1, T_2, T_3) \wedge Happens(S_2, T_4, T_5) \wedge \ldots \wedge \\
&\quad Happens(S_K, T_{2K}, T_{2K+1}) \wedge T_1 < T_2 \wedge T_3 < T_4 \wedge \ldots \wedge \\
&\quad T_{2K-1} < T_{2K} \wedge T_{2K+1} < T_N
\end{aligned}$$

Figure 5-5 Sequence Translation

If-Then-Else

If-Then-Else construct contains two component processes and a condition. When the condition is satisfied *if-component* is executed; otherwise the *else-component* is executed. Its structure is given in Figure 5-6.

```
If-Then-Else Composite Process<C, V, P, π, S_π, S_¬π>
      C : Composite Process Functor

      V : Set of Inputs
             {V_1, V_2, ..., V_N}

      P : Preconditions
            Conjunction of Literals (P_1 ∧ P_2 ∧ ... ∧ P_M)

      π : If condition
            Conjunction of Literals (π_1 ∧ π_2 ∧ ... ∧ π_K)

      S_π : If condition Sub-Process

      S_¬π : Else condition Sub-Process
```

Figure 5-6 If-Then-Else Composite Process

Two *Happens* axioms are written for both cases as it is given in Figure 5-7. With the help of *NotHoldsAllAt* which is logically the negation of *HoldsAllAt*, the second axiom is executed when the *else-case* holds.

$$\begin{aligned}
&Happens(C, T_1, T_N) \leftarrow HoldsAllAt(P, T_1) \wedge \\
&\quad HoldsAllAt(\pi, T_1) \wedge Happens(S_{\pi}, T_2, T_3) \wedge \\
&\quad T_1 < T_2 \wedge T_3 < T_N \\[1ex]
&Happens(C, T_1, T_N) \leftarrow HoldsAllAt(P, T_1) \wedge \\
&\quad NotHoldsAllAt(\pi, T_1) \wedge Happens(S_{\neg\pi}, T_2, T_3) \wedge \\
&\quad T_1 < T_2 \wedge T_3 < T_N \\[1ex]
&NotHoldsAllAt(\{F_1, F_2, ..., F_P\}, T) \leftrightarrow HoldsAt(\neg F_1, T) \vee \\
&\qquad HoldsAt(\neg F_2, T) \vee ... \vee HoldsAt(\neg F_N, T)
\end{aligned}$$

Figure 5-7 If-Then-Else Translation

Repeat-While and Repeat-Until

Repeat-While and *Repeat-Until* constructs contain one component process and a loop controlling condition. The loop iterates as long as the condition holds for *Repeat-While* and does not hold for *Repeat-Until*. They have a common structure and it is given in Figure 5-8.

```
Repeat-While/Unless Composite Process<C, V, P, π, Sπ>
      C : Composite Process Functor

      V : Set of Inputs
             {V1, V2, ..., VN}

      P : Preconditions
            Conjunction of Literals (P1 ∧ P2 ∧ ... ∧ PM)

      π : Loop condition
            Conjunction of Literals (π1 ∧ π2 ∧ ... ∧ πK)

      Sπ : Loop condition Sub-Process
```

Figure 5-8 Repeat-While/Unless Composite Process

Two *Happens* axioms are written for both states of the condition. The same composite event is triggered recursively as long as the condition permits. The translations for *Repeat-While* and *Repeat-Until* are given in Figure 5-9 and Figure 5-10 respectively.

$$
\begin{aligned}
&\mathit{Happens}(C, T_1, T_N) \leftarrow \mathit{HoldsAllAt}(P, T_1) \wedge \\
&\qquad \mathit{HoldsAllAt}(\pi, T_1) \wedge \mathit{Happens}(S_\pi, T_2, T_3) \wedge \\
&\qquad \mathit{Happens}(C, T_4, T_5) \wedge T_1 < T_2 \wedge T_3 < T_4 \wedge T_5 < T_N \\
&\mathit{Happens}(C, T_1, T_1) \leftarrow \mathit{HoldsAllAt}(P, T_1) \wedge \\
&\qquad \mathit{NotHoldsAllAt}(\pi, T_1)
\end{aligned}
$$

Figure 5-9 Repeat-While Translation

$$
\begin{aligned}
&\mathit{Happens}(C, T_1, T_N) \leftarrow \mathit{HoldsAllAt}(P, T_1) \wedge \\
&\qquad \mathit{NotHoldsAllAt}\ (\pi, T_1) \wedge \mathit{Happens}(S_\pi, T_2, T_3) \wedge \\
&\qquad \mathit{Happens}(C, T_4, T_5) \wedge T_1 < T_2 \wedge T_3 < T_4 \wedge T_5 < T_N \\
&\mathit{Happens}(C, T_1, T_1) \leftarrow \mathit{HoldsAllAt}(P, T_1) \wedge \\
&\qquad \mathit{HoldsAllAt}(\pi, T_1)
\end{aligned}
$$

Figure 5-10 Repeat-Until Translation

Any-Order

It is similar to the *Sequence* construct in a sense that the entire component processes are executed consecutively however the ordering can be any permutation of the component set. It has the same specification with the *Sequence* construct with a difference that sub-processes do not constitute an ordered set. The translation requires a permutation predicate which computes permutations of the events. With the help of backtracking different permutations can be tried during execution. The translation is given in Figure 5-11.

$$\begin{aligned}&Happens(C, T_1, T_N) \leftarrow HoldsAllAt(P, T_1) \wedge \\ &\quad Permute(\{S_1, S_2, ..., S_K\}, \{E_1, E_2, ..., E_K\}) \wedge \\ &\quad Happens(E_1, T_2, T_3) \wedge Happens(E_2, T_4, T_5) \wedge ... \wedge \\ &\quad Happens(E_K, T_{2K}, T_{2K+1}) \wedge T_1 < T_2 \wedge T_3 < T_4 \wedge ... \wedge \\ &\quad T_{2K-1} < T_{2K} \wedge T_{2K+1} < T_N\end{aligned}$$

Figure 5-11 Any-Order Translation

Choice

This construct contains a set of component processes and any of them may be chosen for execution. Its structure is exactly the same as the *Any-Order* construct and its translation is given in Figure 5-12.

$$\begin{aligned}&Happens(C, T_1, T_N) \leftarrow HoldsAllAt(P, T_1) \wedge \\ &\quad [Happens(S_1, T_2, T_3) \vee Happens(S_2, T_2, T_3) \vee ... \vee \\ &\quad Happens(S_K, T_2, T_3)] \wedge T_1 < T_2 \wedge T_3 < T_N\end{aligned}$$

Figure 5-12 Choice Translation

Split

Split construct defines a set of component processes which are executed concurrently. It terminates when all of its sub-processes are scheduled to be executed. Its structure is exactly the same as *Any-Order* construct. The translation to the Event Calculus is given in Figure 5-13.

$$
\begin{aligned}
&Happens(C, T_1) \leftarrow HoldsAllAt(P, T_1) \wedge \\
&\quad Happens(S_1, T_2, T_3) \wedge Happens(S_2, T_4, T_5) \wedge \ldots \wedge \\
&\quad Happens(S_K, T_{2K}, T_{2K+1}) \wedge \\
&\qquad T_1 < T_2 \wedge T_1 < T_4 \wedge \ldots \wedge T_1 < T_{2K} \wedge
\end{aligned}
$$

Figure 5-13 Split Translation

A split translation is an instantaneous compound event[7] since component processes are initiated after it has been executed.

Split+Join

Split+Join is a synchronization construct which is used for ensuring that the component processes are completed concurrently within the time segment of the composite process. In this construct, the component processes are initiated and terminated inside the time segment of the containing composite process. Its translation to the Event Calculus is given in Figure 5-14.

[7] Logically it is not meaningful that a compound event is instantaneous and the sub-events are executed after it. However considering the concept of *Split* construct, we thought that it is the best way for the translation.

$$
\begin{aligned}
&Happens(C, T_1, T_N) \leftarrow HoldsAllAt(P, T_1) \wedge \\
&\quad Happens(S_1, T_2, T_3) \wedge Happens(S_2, T_4, T_5) \wedge \ldots \wedge \\
&\quad Happens(S_K, T_{2K}, T_{2K+1}) \wedge \\
&\qquad T_1 < T_2 \wedge T_1 < T_4 \wedge \ldots \wedge T_1 < T_{2K} \wedge \\
&\qquad T_3 < T_N \wedge T_5 < T_N \wedge \ldots \wedge T_{2K+1} < T_N
\end{aligned}
$$

Figure 5-14 Split+Join Translation

5.1.3 Algorithm For OWL-S Translation

The translation of processes to the Event Calculus is a straightforward conversion algorithm which replaces the OWL-S process definitions with the Event Calculus axioms. In fact, each such kind of translation from OWL-S to the composition execution domain requires a variant of the translation which is given in [43]. Hence the conversion algorithm specified in Figure 5-15 is a variant of it.

```
Procedure Convert OWL-S to EC
      Input M : OWL Process Model
      Output T : Event Calculus Theory

      While M ≠ ∅
            Get P ∈ M | P is a Process

            If P = Atomic Process
                  Ax ⇐ Translate Atomic Process P
            Else If P = Composite Sequence Process
                  Ax ⇐ Translate Sequence Process P
            Else If P = Composite If-Then-Else Process
                  Ax ⇐ Translate If-Then-Else Process P
            Else If P = Repeat-While Process
                  Ax ⇐ Translate Repeat-While Process P
            Else If P = Repeat-Until Process
                  Ax ⇐ Translate Repeat-Until Process P
            Else If P = Any-Order Process
                  Ax ⇐ Translate Any-Order Process P
            Else If P = Choice Process
                  Ax ⇐ Translate Choice Process P
            Else If P = Split Process
                  Ax ⇐ Translate Split Process P
            Else If P = Split+Join Process
                  Ax ⇐ Translate Split+Join Process P
            End If

            T ⇐ T ∪ Ax | Ax is the translated axiom set
            M ⇐ M - {P}
      End While
```

Figure 5-15 OWL-S Process Model Translation to the Event Calculus

5.1.4 About The Translation Algorithm

The translation algorithm works on Process Models which does not contain Atomic Processes with conditional outputs since there is no conversion specified for them. They can be added with the help of separate domain axioms which asserts and retracts predicates that are not fluents.

The translations of loop constructs inside the conversion algorithm might generate a theory which might not produce the desired results if stuck in an infinite loop during execution. If the answers are in different branches from the looped one, those branches might never be taken because of the never ending loop in the chosen branch. In such cases completeness of the generated theory is violated because of the inadequacy to generate the expected results. In order to avoid this problem in a practical implementation, an upper-bound for the number of iterations are required. This technique is used in [23] for similar purposes.

Other than the loops given in this translation algorithm there is one more loop composite construct which is *Iterate*. It is a non-deterministic construct since there is no loop controlling condition for this construct. Normally *Iteration* is assumed to be terminated whenever the executer decides, anticipating the necessary conditions ahead of the execution point. The Event Calculus implementation (ATP) we used for execution has no such capability to provide this decision therefore the translation of this construct is not given.

Another constraint is required for the external Web Service calls. They are assumed to be terminable with a consistent result. This requirement is necessary to guarantee the completion of execution with desired effects. This assumption is a general and a common assumption which is required for all such kind of executers which has been first stated in [43].

An example translation of the OWL-S constructs for ATP is given in APPENDIX C. It is a Prolog program for ATP interpretation for the various constructs demonstrated in this section.

5.2 Generic Composition Execution

Generic composition is another method for Web Service Composition which first appeared in [24]. In this technique, Web Service Compositions are prepared manually in GOLOG for execution as it is described in Section 2.4.4.3. In this section the implementation of the traveling problem given in [24] is formulated in the Event Calculus in order to compare it with GOLOG.

5.2.1 An Example Generic Composition

In [24], a generic composition (or procedure in GOLOG) is proposed for the traveling arrangement task. In this procedure, the transportation and hotel booking are arranged and then mail is sent to the customer. Finally an online expense claim is updated.

The transportation via air is selected with the constraint that it should be below the customer's specified maximum price. If the destination is close enough to drive by car then instead of air transportation, car rental is preferred. The customer specifies a maximum drive time for this purpose. If the air transportation is selected then a car is arranged for local transportation. Also a hotel is booked for residence at the destination.

5.2.2 The Event Calculus Implementation

Compound events are used for writing generic compositions in the Event Calculus in a similar way that they have been used in OWL-S translation. The whole operation is decomposed into smaller tasks which are separately captured with other compound events. The Event Calculus translation is given in Figure 5-16.

```
$Happens(Travel(O, D, D_1, D_2), T_1, T_N) \leftarrow$
      $[[Happens(BookFlight(O, D, D_1, D_2), T_2, T_3) \wedge$
      $Happens(BookCar(D, D, D_1, D_2), T_4, T_5) \wedge T_3 < T_4] \vee$
      $Happens(BookCar(O, D, D_1, D_2), T_2, T_5)] \wedge$
      $Happens(BookHotel(D, D_1, D_2), T_6, T_7) \wedge$
      $Happens(SendEmail, T_8) \wedge$
      $Happens(UpdateExpenseClaim, T_9) \wedge$
      $T_5 < T_6 \wedge T_7 < T_8 \wedge T_8 < T_9 \wedge T_9 < T_N$

$Happens(BookFlight(O, D, D_1, D_2), T_1, T_N) \leftarrow$
      $Ex\_GetDriveTime(O, D, Tm) \wedge$
      $Tm > UserMaxDriveTime \wedge$
      $Ex\_SearchForFlight(O, D, D_1, D_2, Id) \wedge$
      $Ex\_GetFlightQuote(Id, Pr) \wedge$
      $Pr < UserMaxPrice \wedge$
      $Ex\_BookFlight(Id)$

$Happens(BookCar(O, D, D_1, D_2), T_1, T_N) \leftarrow$
      $[[Ex\_GetDriveTime(O, D, Tm) \wedge$
      $Tm < UserMaxDriveTime] \vee O = D] \wedge$
      $Ex\_BookCar(O, D, D_1, D_2)$

Where O : Origin, D : Destination, $D_1$ : Traveling Start
      Date, $D_2$ : Traveling End Date
```

Figure 5-16 Generic Composition in the Event Calculus

In this translation *UserMaxDriveTime* and *UserMaxPrice* are the user preference values which alter the flow of operations. Based on traveling inputs and user preferences the traveling arrangement is accomplished with the help of external Web Service calls (in Figure 5-16 they are represented with predicates with *Ex_* prefix).

5.3 Comparison with other Web Service Executers

5.3.1 Comparison with GOLOG

The most important difference between the Event Calculus and GOLOG in the scope of Generic Service Composition is the syntax of both languages. GOLOG provides extralogical constructs which ease the definition of the problem space as it is given in [24] for the same example above. Even though these constructs can be easily covered with primitive Event Calculus axioms; they are not as readable as their GOLOG equivalents. For that reason it can be claimed that GOLOG is a more suitable language to define generic compositions. However it can be concluded that it is not more expressive than the Event Calculus since all of the GOLOG constructs, which constitute a subset of the OWL-S constructs, can be expressed with the Event Calculus.

5.3.2 Comparison with SHOP2

In [43], SHOP2 is used to translate DAML-S process model into SHOP2 operators. This translation assumes certain constraints for the process model to be converted.

The first assumption in SHOP2 translation is that the atomic processes are assumed to be either output generating or effect generating but not both. Atomic processes with conditional effects and outputs are not converted at all. Our translation supports atomic processes with outputs, effects and conditional effects.

Another limitation of SHOP2 translation is the support for concurrent processes. Since SHOP2 cannot handle parallelism the composite constructs *Split* and *Split+Join* cannot be translated. On the other hand, our translation supports for these constructs since Event Calculus is inherently capable of handling concurrency.

CHAPTER VI

CONCLUSION AND FUTURE WORK

Web Services are used throughout the world in distributed web applications. The communication and integration standards for them made it easy to combine different Web Services and accomplish the desired tasks. As the business technologies and customer services are carried to web, the importance of Web Services increased, however the scale of applications, and the complicated and dynamic requests of the clients made it difficult to integrate or compose services manually to serve for the demands. For that reason, automated composition techniques are being developed to solve this problem.

In order to automate the Web Service Composition, the industry and academic society developed Web Service standards, BPEL4WS, OWL-S which are based on previous standards, where WSDL, RDF PDDL are the important ones. In these efforts, the main motivations are to standardize the complicated interactions of a group of Web Services and express the semantics of Web Services in a machine understandable and unambiguous way. With the help of these standards, automated agents might search, combine and execute Web Services that meet the user demands. Since these standards are relatively new, the industry has not fully adopted them yet but in the future it is believed that they will be exploited heavily.

In the literature, planning algorithms have been used to compose Web Services. The main reason behind this preference is the ability to express Web Services as actions of generated plans. Likewise actions, Web Services have parameters, preconditions, results and effects hence they are very attractive to be used in conventional planning algorithms. In order to extract this information the new standards like OWL-S and sample application spaces which are expressed in these standards are used.

In this thesis, the Event Calculus, which is a logical formalism for the description of actions and their effects in dynamic environments, has been used for the solution of Web Services Composition problem. In Event Calculus events and fluents model the dynamically changing world which suits well for the Automated Web Service Composition problem.

It is shown that when a goal situation is given, the Event Calculus can find proper plans as Web Service Compositions with the use of abduction technique. It is shown that the solutions that are generated by the Event Calculus can be compiled into a workflow like composition for the satisfaction of the goal situation. The planning capabilities of the Event Calculus are tested with the PDDL problems with the help of a translation scheme.

Another contribution of this thesis is that the Event Calculus is shown to be utilizable for the execution of OWL-S and generic compositions. In generic compositions Web Services are already integrated to each other however the conditions that are generated by the information collected from the external world during the execution, control the flow of execution. The Event Calculus is used as an execution middle ground between the compositions and Web Services and it is shown that it effectively generates the desired solutions. The Event Calculus is compared with other formalisms which are used for the same purpose. The most important advantage of the Event Calculus is the concurrent execution ability due to timed events.

As a future work, the results that are theoretically expressed and practically captured within examples in this thesis can be put into action and implemented within a system which works in real environments. It would be helpful if common structures of the Web Service compositions are translated into *meta* Event Calculus constructs.

Another improvement might be on queries which are known a priori for the compositions. The queries can be collected from customers in a plain language and then translated into the Event Calculus goals with the help of Natural Language Processing.

REFERENCES

[1] Benjamins V. R., Plaza E., Motta E., Fensel D., Studer R., Wielinga B., Schreiber G., Zdrahal Z., Decker S. IBROW3: An Intelligent Brokering Service for Knowledge-Component Reuse on the World Wide Web. In The Eleventh Banff Knowledge Acquisition for Knowledge-Based System Workshop, 1998.

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

[3] Business Process Execution Language for Web Services Version 1.1, May 2003. Publish of BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG, Siebel Systems. Last Accessed: 17 September 2005. http://www-128.ibm.com/developerworks/library/specification/ws-bpel/

[4] Cicekli N. K., Yildirim Y. Formalizing Workflows Using the Event Calculus. Proceedings of the Eleventh International Conference on Database and Expert Systems Applications, pp. 222--231, 2000.

[5] DAML+OIL Ontology Markup Language, March 2001. Publish of DAML+OIL Joint Committee. Last Accessed: 17 September 2005. http://www.daml.org/2001/03/reference

[6] DAML-S: Semantic Markup for Web Services Version 0.9, May 2003. Publish of The DAML Services Coalition. Last Accessed: 17 September 2005. http://www.daml.org/services/daml-s/0.9/daml-s.html

[7] ebXML Technical Architecture Specification v1.0.4, 16 February 2001. Publish of ebXML Technical Architecture Project Team. Last Accessed: 17 September 2005. http://www.ebxml.org/specs/ebTA.pdf

[8] Erol K., Hendler J., Nau D. UMCP: A sound and complete procedure for hierarchical task network planning. In Proceedings of Second International Conference on AI Planning Systems, pp. 249--254, June 1994.

[9] Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation, 04 February 2004. Publish of XML Core Working Group. Last Accessed: 17 September 2005. http://www.w3.org/TR/2004/REC-xml-20040204/

[10] Fikes R. E., Nilsson N. STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence, Vol. 5(2), pp. 189--208, 1971.

[11] Hull R., Hill M., Berardi D. Semantic Web Services Usage Scenario: e-Service Composition in a Behavior based Framework. Publish of Semantic Web Services Initiative Language Committee. Last Accessed: 17 September 2005. http://www.daml.org/services/use-cases/language/

[12] Interface 1: Process Definition Interchange Process Model, 19 October 1999. Publish of Workflow Management Coalition. Last Accessed: 17 September 2005. http://www.wfmc.org/standards/docs/TC-1016-P_v11_IF1_Process_definition_Interchange.pdf

[13] Kartha G. N., Lifschitz, V. Actions with indirect effects (preliminary report). In Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning, pp. 341--350, 1994.

[14] Knoblock C.A., Minton S., Ambite J. L., Muslea M., Oh J., Frank M. Mixed-initiative, multi-source information assistants. In Proceedings of the WWW Conference, ACM Press, pp. 697--707, May 2001.

[15] Koksal P., Cicekli N. K., Toroslu I. H. Specification of Workflow Processes Using the Action Description Language C. January 2001. Last Accessed: 17 September 2005. http://www.ceng.metu.edu.tr/~nihan/pub/aaai-last.ps

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

[17] Levesque H., Reiter R., Lesperance Y., Lin F., Scherl R. GOLOG: A Logic programming language for dynamic domains. Journal of Logic Programming, Vol. 31(1-3) pp. 59--84, April/June 1997.

[18] MacCarthy J., Hayes P. Some philosophical problems from the standpoint of artificial intelligence. Machine Intelligence Vol. 4, Edinburgh University Press, pp. 463--502, 1969.

[19] Mandell D. J., McIlraith, S. A. Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. The Proceedings of the Second International Semantic Web Conference, 2003.

[20] Martin D., Solanki M. OWL-S 1.0 Release Examples: Bravo Air (fictitious airline site), 03 September 2004. Publish of OWL-S Coalition. Last Accessed: 17 September 2005. http://www.daml.org/services/owl-s/1.0/BravoAirProcess.owl

[21] McDermott D. Estimated-regression planning for interactions with Web Services. In Sixth International Conference on AI Planning and Scheduling. AAAI Press, 2002.

[22] McDermott D. V., Dou D., Qi P. PDDAML, An Automatic Translator Between PDDL and DAML. Last Accessed: 17 September 2005. http://www.cs.yale.edu/homes/dvm/daml/pddl_daml_translator1.html

[23] McIllraith S. A., Fadel R. Planning with Complex Actions. Proceedings of the Ninth International Workshop on Non-Monotonic Reasoning, pp. 356--364, April 2002.

[24] McIlraith S. A., Son T. Adapting Golog for composition of semantic Web services. In Proceedings of Eight International Conference on Principles of Knowledge Representation and Reasoning, pp. 482--493, 2002.

[25] McIlraith S. A., Son T., and Zeng H. Semantic Web services. In IEEE Intelligent Systems, March/April 2001.

[26] Medjahed B., Bouguettaya A., Elmagarmid A. K. Composing web services on the semantic web. The VLDB Journal, Vol. 12(4), pp. 333--351, November 2003.

[27] OWL-S: Semantic Markup for Web Services Version 1.1, November 2004. Publish of Semantics Web Services Language (SWSL) Committee. Last Accessed: 17 September 2005. http://www.daml.org/services/owl-s/1.1/overview/

[28] Pednault E. ADL: Exploring the middle ground between STRIPS and the situation calculus. In Proceedings of First International Conference on Principles of Knowledge Representation and Reasoning, pp. 324--332, 1989.

[29] Reiter R. Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press, 2001.

[30] Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C Recommendation, 10 February 2004. Publish of RDF Core Working Group. Last Accessed: 17 September 2005. http://www.w3.org/TR/rdf-concepts/

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

[32] Shanahan M.P. An abductive event calculus planner. Journal of Logic Programming, Vol. 44(1-3), pp. 207--240, July 2000.

[33] 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.

[34] Srivastava, B. SWORD: A Developer Toolkit for Web Service Composition. In Proceedings of the Eleventh International World Wide Web Conference, 2002.

[35] Su X., Rao J. A Survey of Automated Web Service Composition Methods. In Proceedings of First International Workshop on Semantic Web Services and Web Process Composition, July 2004.

[36] The Planning Domain Definition Language Version 1.2, October 1998. Publish of Artificial Intelligence Planning Systems Planning Competition Committee. Last Accessed: 17 September 2005. ftp://ftp.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz

[37] Tsur S., Abiteboul S., Agrawal R., Dayal U., Klein J., Weikum G. Are Web services the next revolution in e-commerce?. In Proceedings of the Twenty Seventh VLDB conference, pp. 614--617, 2001.

[38] UDDI Version 3.0.2 UDDI Spec Technical Committee Draft, 19 November 2004. Publish of UDDI Technical Committee. Last Accessed: 17 September 2005. http://uddi.org/pubs/uddi_v3.htm

[39] Veloso M. Planning, Execution, and Learning Course Notes: PDDL by Example. Carnegie Mellon University, Fall 2002. Last Accessed: 08 August 2005. http://www.cs.cmu.edu/~mmv/planning/homework/PDDL_Examples.pdf

[40] Verma K., Sheth A., Miller J., Aggarwal R. Semantic Web Services Usage Scenario: Dynamic QoS based Supply Chain. Publish of Semantic Web Services Initiative Architecture Committee. Last Accessed: 17 September 2005. http://www.daml.org/services/use-cases/architecture/

[41] Waldinger R. Deductive composition of Web software agents. In Proceedings of NASA Workshop on Formal Approaches to Agent-Based Systems, LNCS 1871, Springer-Verlag, 2000.

[42] Web Services Description Language (WSDL) 1.1, W3C Note, 15 March 2001. Publish of Ariba, International Business Machines Corporation, Microsoft. Last Accessed: 17 September 2005. http://www.w3.org/TR/wsdl

[43] Wu D., Sirin E., Parsia B, Hendler J., Nau D. Automatic web services composition using SHOP2. In Proceedings of Planning for Web Services Workshop, in ICAPS 2003, June 2003.

APPENDIX A

A WSDL sample document for stock quote operations is given below.

```
<?xml version="1.0"?>
<definitions name="StockQuote"
             targetNamespace="http://example.com/stockquote.wsdl"
             xmlns:tns="http://example.com/stockquote.wsdl"
             xmlns:xsd1="http://example.com/stockquote.xsd"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <schema targetNamespace="http://example.com/stockquote.xsd"
            xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="TradePriceRequest">
        <complexType>
          <all>
            <element name="tickerSymbol" type="string"/>
          </all>
        </complexType>
      </element>
      <element name="TradePrice">
         <complexType>
           <all>
             <element name="price" type="float"/>
           </all>
         </complexType>
      </element>
    </schema>
  </types>

  <message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
  </message>
  <message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
  </message>

  <portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
```

```
      <input message="tns:GetLastTradePriceInput"/>
      <output message="tns:GetLastTradePriceOutput"/>
    </operation>
  </portType>

  <binding name="StockQuoteSoapBinding"
           type="tns:StockQuotePortType">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetLastTradePrice">
      <soap:operation
            soapAction="http://example.com/GetLastTradePrice"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>

  <service name="StockQuoteService">
    <documentation>My first service</documentation>
    <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
      <soap:address location="http://example.com/stockquote"/>
    </port>
  </service>
</definitions>
```

APPENDIX B

An example OWL-S Composite Construct Sequence is given below. This example is adopted from [20]. Default sections that contain identification, imports and definitions are omitted for simplicity. For full specification refer to [20].

```
<!-- ####################################################### -->
<!-- Instance Definition of BravoAir Reservation Agent Process
     Model -->

<process:ProcessModel
      rdf:ID="BravoAir_ReservationAgent_ProcessModel">
  <process:hasProcess rdf:resource="#BravoAir_Process" />
  <service:describes
      rdf:resource="&ba_service;#BravoAir_ReservationAgent"/>
</process:ProcessModel>

<!-- ####################################################### -->
<!-- Definition of top level Process as a composite process -->
<!-- BravoAir_Process is a composite process. It is composed of
     a sequence whose components are 2 atomic processes,
     GetDesiredFlightDetails and SelectAvailableFlight, and a
     composite process, BookFlight. -->

<process:CompositeProcess rdf:ID="BravoAir_Process">
  <rdfs:label>
    This is the top level process for BravoAir
  </rdfs:label>
  <process:composedOf>
    <process:Sequence>
      <process:components rdf:parseType="Collection">
      <process:AtomicProcess
          rdf:about="#GetDesiredFlightDetails"/>
      <process:AtomicProcess
          rdf:about="#SelectAvailableFlight"/>
      <process:CompositeProcess
          rdf:about="#BookFlight"/>
      </process:components>
    </process:Sequence>
  </process:composedOf>
</process:CompositeProcess>

<!-- ######################################################## -->
<!-- BookFlight (Composite) Log into account and confirm
     reservation -->
<!-- BookFlight is a composite process. It is composed of a
     sequence whose components are 2 atomic processes, LogIn and
     ConfirmReservation. -->

 <process:CompositeProcess rdf:ID="BookFlight">
  <process:composedOf>
    <process:Sequence>
      <process:components rdf:parseType="Collection">
        <process:AtomicProcess
            rdf:about="#Login"/>
        <process:AtomicProcess
            rdf:about="#ConfirmReservation"/>
      </process:components>
    </process:Sequence>
  </process:composedOf>
</process:CompositeProcess>

<!-- ######################################################## -->
<!-- GetDesiredFlightDetails (ATOMIC) Get details such as
     airports,  prefered time, roundtrip etc -->

<process:AtomicProcess rdf:ID="GetDesiredFlightDetails">
  <process:hasInput rdf:resource="#DepartureAirport_In"/>
  <process:hasInput rdf:resource="#ArrivalAirport_In"/>
  <process:hasInput rdf:resource="#OutboundDate_In"/>
  <process:hasInput rdf:resource="#InboundDate_In"/>
  <process:hasInput rdf:resource="#RoundTrip_In"/>
</process:AtomicProcess>

<process:Input rdf:ID="DepartureAirport_In">
  <process:parameterType rdf:resource="&concepts;#Airport"/>
</process:Input>

<process:Input rdf:ID="ArrivalAirport_In">
  <process:parameterType rdf:resource="&concepts;#Airport"/>
</process:Input>

<process:Input rdf:ID="OutboundDate_In">
  <process:parameterType rdf:resource="&concepts;#FlightDate"/>
</process:Input>

<process:Input rdf:ID="InboundDate_In">
  <process:parameterType rdf:resource="&concepts;#FlightDate"/>
</process:Input>
```

```
<process:Input rdf:ID="RoundTrip_In">
  <process:parameterType rdf:resource="&concepts;#RoundTrip"/>
</process:Input>

<!-- ######################################################## -->
<!-- SelectAvailableFlight (ATOMIC) Get users prefered flight
     choice from available itineraries -->

<process:AtomicProcess rdf:ID="SelectAvailableFlight">
  <process:hasInput
      rdf:resource="#PreferredFlightItinerary_In"/>
  <process:hasOutput
      rdf:resource="#AvailableFlightItineraryList_Out"/>
</process:AtomicProcess>

<process:Input rdf:ID="PreferredFlightItinerary_In">
  <process:parameterType
      rdf:resource="&concepts;#FlightItinerary"/>
</process:Input>

<process:UnConditionalOutput
    rdf:ID="AvailableFlightItineraryList_Out">
  <process:parameterType
      rdf:resource="&concepts;#FlightItineraryList"/>
</process:UnConditionalOutput>

<!-- ######################################################## -->
<!-- LogIn (ATOMIC) Get user details -->

<process:AtomicProcess rdf:ID="LogIn">
  <process:hasInput rdf:resource="#AcctName_In"/>
  <process:hasInput rdf:resource="#Password_In"/>
</process:AtomicProcess>

<process:Input rdf:ID="AcctName_In">
  <process:parameterType rdf:resource="&concepts;#AcctName"/>
</process:Input>
<process:Input rdf:ID="Password_In">
  <process:parameterType rdf:resource="&concepts;#Password"/>
</process:Input>

<!-- ######################################################## -->
<!-- ConfirmReservation (ATOMIC) Confirm selected reservation -->

<process:AtomicProcess rdf:ID="ConfirmReservation">
  <process:hasInput rdf:resource="#ReservationID_In"/>
  <process:hasInput rdf:resource="#Confirm_In"/>
  <process:hasOutput
      rdf:resource="#PreferredFlightItinerary_Out"/>
  <process:hasOutput rdf:resource="#AcctName_Out"/>
  <process:hasOutput rdf:resource="#ReservationID_Out"/>
  <process:hasEffect rdf:resource="#HaveSeat"/>
</process:AtomicProcess>

<process:Input rdf:ID="ReservationID_In">
  <process:parameterType
      rdf:resource="&concepts;#ReservationNumber"/>
</process:Input>

<process:Input rdf:ID="Confirm_In">
  <process:parameterType rdf:resource="&concepts;#Confirmation"/>
</process:Input>

<process:UnConditionalOutput
    rdf:ID="PreferredFlightItinerary_Out">
  <process:parameterType
      rdf:resource="&concepts;#FlightItinerary"/>
</process:UnConditionalOutput>

<process:UnConditionalOutput rdf:ID="AcctName_Out">
  <process:parameterType rdf:resource="&concepts;#AcctName"/>
</process:UnConditionalOutput>

<process:UnConditionalOutput rdf:ID="ReservationID_Out">
  <process:parameterType
      rdf:resource="&concepts;#ReservationNumber"/>
</process:UnConditionalOutput>

<process:UnConditionalEffect rdf:ID="HaveSeat">
  <process:ceEffect rdf:resource="&concepts;#HaveFlightSeat"/>
</process:UnConditionalEffect>
```

APPENDIX C

PDDL Translation Example

PDDL translation example given in this thesis is written in Prolog for the interpretation of ATP:

```
/* static axioms */
axiom(room(rooma), []).
axiom(room(roomb), []).
axiom(ball(ball1), []).
axiom(ball(ball2), []).
axiom(gripper(right), []).
axiom(gripper(left), []).

/* fluents */
axiom(initially(at-robby(rooma)), []).

axiom(initially(free(right)), []).
axiom(initially(free(left)), []).

axiom(initially(at-ball(ball1, rooma)), []).
axiom(initially(at-ball(ball2, rooma)), []).

/* actions */
axiom(initiates(move(From, To), at-robby(To), T),
      [room(From), room(To), diff(From, To),
      holds_at(at-robby(From), T)]).
axiom(releases(move(From, To), at-robby(From), T),
      [room(From), room(To), diff(From, To),
      holds_at(at-robby(From), T)]).

axiom(initiates(pick-up(B, R, G), carry(G, B), T),
      [ball(B), room(R), gripper(G), holds_at(at-robby(R), T),
      holds_at(at-ball(B, R), T), holds_at(free(G), T)]).
axiom(releases(pick-up(B, R, G), at-ball(B, R), T),
      [ball(B), room(R), gripper(G), holds_at(at-robby(R), T),
      holds_at(at-ball(B, R), T), holds_at(free(G), T)]).
axiom(releases(pick-up(B, R, G), free(G), T),
      [ball(B), room(R), gripper(G), holds_at(at-robby(R), T),
      holds_at(at-ball(B, R), T), holds_at(free(G), T)]).

axiom(initiates(drop(B, R, G), at-ball(B, R), T),
      [ball(B), room(R), gripper(G), holds_at(carry(G, B), T),
      holds_at(at-robby(R), T)]).
axiom(initiates(drop(B, R, G), free(G), T),
      [ball(B), room(R), gripper(G), holds_at(carry(G, B), T),
      holds_at(at-robby(R), T)]).
axiom(releases(drop(B, R, G), carry(G, B), T),
      [ball(B), room(R), gripper(G), holds_at(carry(G, B), T),
      holds_at(at-robby(R), T)]).

/* executable actions */
executable(move(From, To)).
executable(pick-up-both(B1, B2, R, G1, G2)).
executable(pick-up(B, R, G)).
executable(drop(B, R, G)).

/* abduction policy */
abducible(dummy).

/* sample queries */
abdemo([holds_at(carry(left, ball1), t)], R).
abdemo([holds_at(at-ball(ball1, roomb), t)], R).
abdemo([holds_at(at-ball(ball2, roomb), t)], R).
abdemo([holds_at(at-ball(ball1, roomb), t),
      holds_at(at-ball(ball2, roomb), t)], R).
abdemo([holds_at(at-ball(ball1, roomb), t),
      holds_at(at-robby(roomb), t)], R).
```

OWL-S Translation Example

OWL-S translation example which is written in Prolog for interpretation of ATP is given below:

```
executable(atomic1).
executable(atomic2).

%%%%%%%%%% SEQUENCE %%%%%%%%%%

axiom(initially(sequencePreCond), []).
%> axiom(initially(neg(sequencePreCond)), []).

axiom( happens( sequence, T0, TN ),
            [ holds_at(sequencePreCond, T0),
              happens( atomic1, T1, T2 ),
              happens( atomic2, T3, T4 ),
              before( T0, T1 ),
              before( T2, T3 ),
              before( T4, TN ) ] ).

% abdemo([happens(sequence, _, _)], R).

%%%%%%%%%% AND-SPLIT %%%%%%%%%%

axiom(initially(andSplitPreCond), []).
%> axiom(initially(neg(andSplitPreCond)), []).

axiom( happens( andSplit, T0, TN ),
            [ holds_at(andSplitPreCond, T0),
              happens( atomic1, T11, T12 ),
              happens( atomic2, T21, T22 ),
              before(T0, T11),
              before(T0, T21),
              before(T12, TN),
              before(T22, TN) ] ).

%%%%%%%%%% AND-JOIN %%%%%%%%%%

axiom(initially(andJoinPreCond), []).
%> axiom(initially(neg(andSplitPreCond)), []).

axiom( happens( andJoin, T0, TN ),
            [ holds_at(andJoinPreCond, T0),
              happened( atomic1, T11, T12 ),
              happened( atomic2, T21, T22 ),
              before(T12, T0),
              before(T21, T0) ] ).

axiom( happens( andSplitJoin, T0, TN ),
            [ happens( andSplit, T1, T2 ),
              happens( andJoin, T3, T4),
              before( T0, T1 ),
              before( T2, T3 ),
              before( T4, TN ) ]).

% abdemo([happens(andSplitJoin, _, _)], R).

%%%%%%%%%% IF-THEN-ELSE %%%%%%%%%%

axiom(initially(ifThenElsePreCond), []).
%> axiom(initially(ifThenElseCond), []).
axiom(initially(neg(ifThenElseCond)), []).

axiom( happens( ifThenElse, T0, TN ),
            [ holds_at(ifThenElsePreCond, T0),
              holds_at(ifThenElseCond, T0),
              happens( atomic1, T1, T2 ),
              before( T0, T1 ),
              before( T2, TN ) ] ).

axiom( happens( ifThenElse, T0, TN ),
            [ holds_at(ifThenElsePreCond, T0),
              holds_at(neg(ifThenElseCond), T0),
              happens( atomic2, T1, T2 ),
              before( T0, T1 ),
              before( T2, TN ) ] ).

% abdemo([happens(ifThenElse, _, _)], R).

%%%%%%%%%% CHOICE %%%%%%%%%%

axiom(initially(choicePreCond), []).
%> axiom(initially(neg(choicePreCond)), []).

axiom( happens( choice, T0, TN ),
            [ holds_at(choicePreCond, T0),
              happens( atomic1, T1, T2 ),
              before( T0, T1 ),
              before( T2, TN ) ] ).

axiom( happens( choice, T0, TN ),
            [ holds_at(choicePreCond, T0),
              happens( atomic2, T1, T2 ),
              before( T0, T1 ),
              before( T2, TN ) ] ).

% abdemo([happens(choice, _, _)], R).

%%%%%%%%%% ANY-ORDER %%%%%%%%%%

axiom(initially(anyOrderPreCond), []).
%> axiom(initially(neg(anyOrderPreCond)), []).

axiom( happens( anyOrder, T0, TN ),
            [ holds_at(anyOrderPreCond, T0),
              happens( atomic1, T1, T2 ),
              happens( atomic2, T3, T4 ),
              before( T0, T1 ),
              before( T2, T3 ),
              before( T4, TN ) ] ).

axiom( happens( anyOrder, T0, TN ),
            [ holds_at(anyOrderPreCond, T0),
              happens( atomic2, T1, T2 ),
              happens( atomic1, T3, T4 ),
              before( T0, T1 ),
              before( T2, T3 ),
              before( T4, TN ) ] ).
```

```
% abdemo([happens(anyOrder, _, _)], R).

%%%%%%%%%% REPEAT-WHILE %%%%%%%%%%

axiom(initially(repeatWhilePreCond), []).
%> axiom(initially(neg(anyOrderPreCond)), []).

axiom( happens( repeatWhile(N, M), T0, TN ),
            [ holds_at(repeatWhilePreCond, T0),
              N < M,
              happens( atomic1, T1, T2 ),
              before( T0, T1 ),
              NP1 is N + 1,
              happens( repeatWhile(NP1, M), T2, T3 ),
              before( T0, T1 ),
              before( T3, TN ) ] ).

axiom( happens( repeatWhile(N, M), T0, _ ),
            [ holds_at(repeatWhilePreCond, T0),
              N >= M ] ).

% abdemo([happens(repeatWhile(1, 3), _, _)], R).
```