Scalable Composition of Web Services Under Uncertainty

by

Haibo Zhao

(Under the direction of Prashant Doshi)

Abstract

Facilitating the assembly of services to form composite services is an important functionality in Service-oriented architecture (SOA). In this dissertation, we focus on the problem of automatically assembling WSs to form compositions that optimize given user preferences. This problem is often referred to as the *automated Web service composition* problem. Prevalent approaches for automatically composing WSs predominantly utilize planning techniques to achieve the composition. However, classical artificial intelligence (AI) planning based approaches are facing the following issues: (1) They are incapable of capturing the uncertainties of Web service behaviors, (2) It is hard for them to provide process optimization during planning, and (3) Many of them are unable to scale efficiently to large processes. To address these issues, we present a hierarchical decision-theoretic planning framework for composing Web services, called Haley. Compared to classical AI planners, the decision-theoretic planning has the ability to capture the uncertainty inherent in WSs and provide a cost based process optimization. Haley uses symbolic planning techniques that operate directly on first order logic based representations of the state space to obtain the compositions. As a result, it supports an automated elicitation of the corresponding planning domain from WS descriptions and produces a compact domain representation in comparison to classical AI planners. Additionally, it tackles the scalability issue by exploiting the hierarchy found in processes. Our experiments demonstrate that Haley evaluates favorably in comparison to other WS

composition approaches. We implement Haley and provide a comprehensive tool suite. The suite accepts WSs described using standard languages such as SAWSDL. It provides process designers with an intuitive interface to specify process requirements, goals and a hierarchical decomposition, and automatically generates BPEL processes, while hiding the complexity of the planning and BPEL from users.

Another emerging research topic is automated REpresentational State Transfer (REST)ful WS composition. While automating WSDL/SOAP WS composition has been extensively studied, automated RESTful WS composition is less explored in the research community. As an early effort addressing this problem, this dissertation discusses the challenges of composing RESTful WSs and proposes a formal model for describing and automatically composing RESTful WSs.

INDEX WORDS: Web service composition, RESTful WSs, decision-theoretic planning, first-order logic, hierarchy

Scalable Composition of Web Services Under Uncertainty

by

Haibo Zhao

B.E., Xiangtan University, China, 2003

A Dissertation Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Doctor of Philosophy

Athens, Georgia

2009

Scalable Composition of Web Services Under Uncertainty

by

Haibo Zhao

Approved:

Major Professor: Prashant Doshi

Committee: John Miller
Khaled Rasheed
Amit Sheth

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August 2009

Dedication

To the love of my family

Acknowledgments

First of all, I would like to acknowledge the contributions of my advisor, Prof. Prashant Doshi, in making this work possible. I would like to gratefully and sincerely thank him for his visionary, guidance, understanding and patience. Not only did he point out some important research directions of the initial ideas, but he also encouraged me to explore deeper issues and challenge harder problems. I have benefited the most from his vision towards my research work. He has been putting my career in his mind and guiding me along the way. I have learned numerous things from him, from presentation skills, experimental methods, paper writing to approaches of conducting research. From the bottom of my heart, I would like to thank him for making my graduate school study an enjoyable and fruitful experience.

I would like to thank the Department of Computer Science, especially the members of my doctoral committee for their input, valuable discussions and accessibility. I would also like to mention my colleagues in the LSDIS lab, John Harney and Yunzhou Wu, who inspired me during our discussions. These two friends and co-workers also provided for some much needed humor and entertainment.

Finally, I would like to thank my parents and family, who have been believing my potential and giving me constant love and support.

This dissertation work was funded by the "Dissertation Completion Award" from the Graduate School, The University of Georgia.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Service-oriented architecture (SOA) aims to provide a loosely coupled integration of services residing on heterogeneous systems, written using different programming languages and with other implementation disparities. Popularly considered as the building blocks of SOA, WSs are self-describing and platform-independent applications that can be invoked over the Web. Facilitating the assembly of services to form composite services is an important functionality in SOA. Users may combine and reuse these services toward the production of business applications. In this dissertation, we focus on the problem of automatically assembling WSs to form compositions that optimize given user preferences. This problem is often referred to as the *Web service composition* problem, although no consensus definition exists.

In this chapter, we introduce two paradigms of WSs – WSDL/SOAP WSs and RESTful WSs – and briefly describe our proposed solutions to the composition problem of these two types of WSs. We summarize our contributions in Section 1.4 and list the structure of the dissertation in Section 1.5

1.1 Web Services

Generally, there are two paradigms of building WS based applications: operation-centric and resource-centric. Operation-centric WSs view the application system from the perspective of operations, that is, what operations to expose in the system. This type of WSs is usually described using Web Service Description Language (WSDL) [25, 26]. The interaction protocol is usually Simple Object Access Protocol (SOAP) [90]. In this dissertation, we call WSs of this type WSDL/SOAP WSs. In contrast, RESTful WSs follow a resource-centric style and

view the application system from the perspective of resources, that is, what resources to expose in the system. We describe the major characteristics and protocols of both kinds of WSs below:

WSDL/SOAP Web Service has been supported by major enterprise computing solution providers such as IBM, SUN, Microsoft and Oracle. It is generally considered to be a "big" [71, 81] and comprehensive solution to system integration in a heterogeneous environment. Its technology stack includes a series of XML based standards and protocols defined by World Wide Web Consortium (W3C) [7] and Organization for the Advancement of Structured Information Standards (OASIS) [4] to ensure inter-operability. The various standards and protocols are:

- Description Protocol: WSDL describes the functionality (public interfaces and available operations) of WSs and the XML messages exchanged between service providers and clients. An interface in WSDL is a set of supported operations, and an operation is defined by the incoming and outgoing XML messages. The message types are usually defined using XML Schema [93]. Services in a WSDL file are defined as a collection of endpoints or ports. Each port is associated with a URI with a binding, where the operations are bounded with a concrete communication protocol (usually SOAP).

- Communication Protocol: SOAP is the communication protocol for exchanging messages between different applications. SOAP is the messaging protocol for WSs. It relies on other application layer protocols (usually HTTP, but other application protocols like FTP, SMTP are also possible) to transmit messages. These application layer protocols are used as options for the underlying transport protocol in SOAP. A SOAP document is basically an XML document with the following information in its top XML level element envelope: (1) An optional Header element contains header information, which includes application-specific information such as authentication, reliability, payment, etc. (2) A Body element contains call and response information, which is the payload of the SOAP message. Similar to WSDL, XML Schema is used to describe the structure

of the SOAP message. XML and XML schema based representation makes SOAP an inter-operable protocol across heterogeneous platforms. (3) An optional Fault element contains errors and status information.

- Publishing and Discovery Protocol: Universal Description, Discovery and Integration (UDDI) [60] is an XML-based registry for service providers to list their services by publishing their descriptions and for users to discover needed WSs. There are three major components in a UDDI business registration: (1)White Pages that list the address and other contact information about the service provider; (2)Yellow Pages that list available WSs by industrial categories based on standard taxonomies; and (3) Green Pages that give detailed technical information about the particular WS, which can usually be populated from WSDL descriptions.

- Extension Protocols: The three protocols above form the foundation of building operation-centric WSDL/SOAP WSs. However, in more complex application environments, other advanced needs (such as atomic transactions, security, trust, reliable messaging, etc.) also have to be accommodated. OASIS has standardized a fairly comprehensive set of WS-* protocols [9]. For example, WS-AtomicTransaction [66] is to support atomic transactions in WS interactions; WS-security [63] protocol is to provide a means for applying security to Web services by using security assertion markup language (SAML) [62], Kerberos [30], and certificate formats such as X.509 [42]. WS-Trust [65] is to enable applications to construct trusted SOAP message exchanges, and WS-ReliableMessaging [61] protocol allows SOAP messages to be reliably delivered between distributed applications.

RESTful Web Service is another alternative way of building WSs. This type of WS is gaining increasing attention in the industry and has been widely adopted by leading Internet companies due to its simplicity and light-weight nature. The key idea of RESTful Web services is to apply REST principles into the development of WSs. First introduced by

Fielding [33, 34, 35], the principles of REpresentational State Transfer (REST) have backed the development of the World Wide Web (WWW). The principles of REST include:

- Resource Centric: Conceptual entities and functionalities are modeled as resources identified by universal resource identifiers (URIs).
- Uniform Interface: Resources are accessed and manipulated via standardized operations (GET, POST, PUT and DELETE) in the HTTP protocol [91]. GET is a side-effect free operation to fetch resource representations; POST, PUT and DELETE are used to create, update and delete resources, respectively.
- State Transfer: Components of the system communicate via these standard interface operations and exchange the representations of these resources (one resource may have multiple representations). In a REST system, servers and clients typically transit through different states of resource representations by following the inter-links between resources.

Due to the different perspectives of building WSs, the composition problem of WSDL/SOAP WSs and RESTful WSs are fundamentally different. We present our proposed solutions to the composition problem of both types of WSs in this dissertation.

In the rest of this chapter, we discuss the motivation and briefly present our proposed solutions in Section 1.2 and Section 1.3. Additionally, we outline our contributions in Section 1.4.

1.2 WSDL/SOAP Web Service Composition

We briefly present in this chapter the motivation and our proposed approach to WSDL/SOAP WS composition: Haley, a hierarchial framework for WS composition at the logical level.

1.2.1 Motivation of Haley

Prevalent approaches for automatically composing WSDL/SOAP WSs predominantly utilize planning techniques to achieve the composition because of the similarity of the Web service composition (WSC) problem to the planning problem in the artificial intelligence (AI) research. While a variety of such approaches have been proposed [45, 53, 56, 68, 73, 75, 78, 88, 99], many of these fail to appropriately identify the differences between the WSC problem and AI planning problem. The following characteristics of WSs and compositions are often not well modeled in existing approaches:

- **Uncertainty:** Distributed computing environments are inherently uncertain: invocation of remote WSs may potentially produce unexpected responses. A response could depend on whether the WS is operating correctly and on external real-world situations. For example, output from an air ticket reservation WS will depend on whether the service is working correctly and whether the airline has tickets remaining. Thus, uncertainty often results from imperfect reliability of WSs and business logic.

- **Optimality:** Although satisfying the functional requirements is important for building the WSC, optimization of nonfunctional preferences may be equally crucial, especially in performance-sensitive application domains. We may wish to build a WSC that minimizes the response times and WS costs, and guarantees a basic level of reliability.

- **Scalability:** The problem size of the composition problem could become very large due to the increase of the number of component services, the choice of WSs to select from and the different types of input that a composition can process. In many cases, the desire for scalability draws a line between "practical" and "impractical" solutions to a specific SOA application problem.

A large number of the proposed approaches for WSC utilize *classical AI planning* [45, 56, 68, 73, 75, 88, 99]. While these approaches guarantee compositions that meet the functional requirements, they are unable to provide the previously mentioned characteristics. In particular, they are:

- **Incapable of modeling uncertainty of WS invocations** Classical planning assumes that the actions, used to model WS invocations, are deterministic. In other words, regardless of the status of the WS or the real-world situation, classical planners assume that the WS will return the expected result. Thus, these approaches often ignore service failures and the possibility of multiple service responses.

- **Failure to provide QoS based optimality** The approaches typically focus on building WSCs that satisfy the functional requirements. Classical planners do not associate cost or rewards with planning states or actions. Therefore, these approaches fail to distinguish between WSs and compositions with identical functionality but exhibiting different quality measurements. This limitation prevents classical planning from optimizing compositions with respect to QoS parameters during planning time or it leads to computational overhead if an additional plan optimization phase is used. In this document, we focus on the WS QoS parameters of cost, reliability and response times.

- **Inability to scale efficiently** The inability to scale is due to an inefficient representation of the WS planning problem and the complexity of the planning algorithms. Many planners use propositional logic to represent the planning domains. Propositional planning is PSPACE-complete, even if operators are restricted to have two positive (non-negated) preconditions and two postconditions, or if operators are restricted to one postcondition (with any number of preconditions). It is NP-complete if operators are restricted to positive postcondition, even if operators are restricted to one precondition and one positive postcondition [20]. Furthermore, handling multiple, different types of input requires additional propositions.

Consequently, we focus on assembling WSs resulting in a composition that is expected to optimize the QoS parameters in the context of uncertainty of the WSs.

In order to address some of these issues, we adopt decision-theoretic planning [16] for composing WSs. Decision-theoretic planners such as Markov decision processes (MDPs) [13] *generalize* classical planning techniques to nondeterministic environments where action outcomes may be uncertain, and associate costs to the different plans thereby allowing the selection of an optimal plan.

In an early piece of work, Doshi et al. [31] showed how we may use MDPs to compose WS-based workflows. Compared to classical planners, a decision-theoretic planner models the uncertainty inherent in WSs using probabilities and facilitates a cost-based process optimization. This approach is especially relevant in the context of SOAs where services may fail and processes must minimize costs. We examined the application of semi-MDP planning toward the hierarchical composition of WSs, and demonstrated that decision-theoretic planning effectively addresses both, the uncertainty and optimality issues outlined previously. As a further improvement of our work along this line, we developed Haley.

1.2.2 Haley: A Hierarchical Framework for Logic Composition of Web Services

To address the automated composition problem of WSDL/SOAP WSs, we propose a novel hierarchical, decision-theoretic planning based framework for automatically composing WSs, which we call Haley. Existing WSC techniques are further plagued by two challenges: (*i*) As the number of participating WSs increases, there is an explosion in the size of the state space; (*ii*) there is a growing consensus among the WS description standards such as OWL-S [51] and SAWSDL [32] on using first order logic (or its variants) to logically represent the preconditions and effects of WSs. However, many of the existing planning techniques used in WS composition do not use the full generality of first order logic while planning. Haley *improves* on previous work by allowing WS composition at the logical level. Specifically,

Haley enables composition using the first order sentences that represent the preconditions and effects of the component WSs. In addition to using a *symbolic representation*, Haley promotes scalability by exploiting possible *hierarchical decomposition* of real-world processes. In many cases, a business process may be seen as nested – a higher level process may be composed of WSs and lower level processes – which induces a natural hierarchy over the process. In order to do this, Haley models each level of the hierarchy using a *first order semi-Markov decision process* (FO-SMDP) that extends a SMDP [74] to operate directly on first order logic sentences, which provide a logical representation of the traditional state space. In particular, the lowest levels of the hierarchy (leaves) are modeled using a FO-SMDP containing *primitive* actions which are invocations of the WSs. Higher levels of the process are modeled using FO-SMDPs that contain *abstract* actions as well, which represent the execution of lower level processes. We represent their invocations as *temporally extended* actions in the higher level FO-SMDPs. Since descriptions of only the individual WSs are usually available, we provide methods for deriving the model parameters of the higher level FO-SMDP from the parameters of the lower level ones. Thus, our approach is applicable to WSCs that are nested to an arbitrary depth.

Haley brings three specific contributions toward composing WSs. First, it offers a way to mitigate the problem of large state spaces by composing at the first order logic level and preserves the expressiveness of first order logic. Consequently, Haley supports an automated elicitation of the corresponding planning domain from WS descriptions and produces a compact domain representation in comparison to classical planning based approaches. Second, Haley offers a way to exploit possible hierarchical decomposition of the WSC problem, thereby further promoting scalability. Third, Haley generates an optimal composition of WSs at each level of the hierarchy. Because parameters of abstract actions are derived from those of the WSs at the lower level, the generated composition is globally optimal under the assumption that the reliabilities of lower level WSs are independent of each other.

Due to the limitations of the existing approaches mentioned above and the complexity of the WSC problem itself, few implemented tools exist. We have implemented Haley[1] and provide a comprehensive tool suite. The suite accepts WSs described using standard languages such as SAWSDL. It provides process designers with an intuitive interface to specify process requirements, goals and a hierarchical decomposition, and automatically generates Web Services Business Process Execution Language (WS-BPEL) [64], while hiding in the processes the complexity of planning and of BPEL from users.

We show a high-level architecture of Haley in Fig. 1.1. Details of the approach are presented in Chapter 4. Our system parses the functional and nonfunctional information from the input WS descriptions files, takes the composition goal and any hierarchy from the process designers, to formulate the WSC problem. The composition problem is then represented as a stochastic planning problem in first order logic and solved using a Prolog based planner. The generated plan is converted into WS-BPEL, which may be deployed in a WS-BPEL implementation.

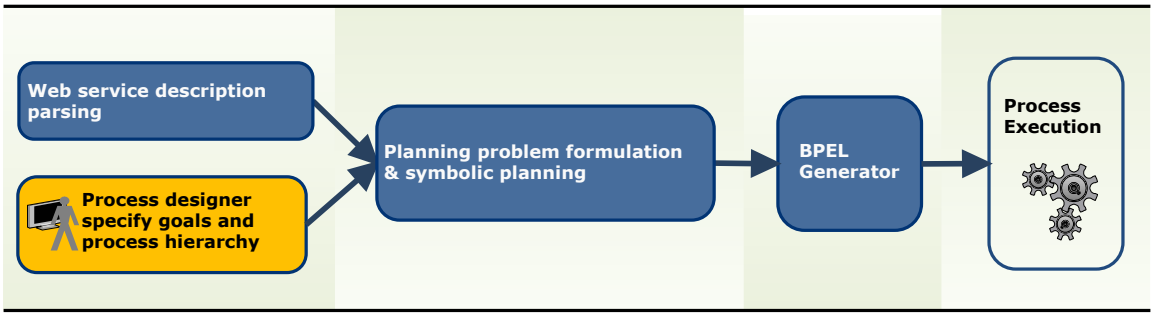

Figure 1.1: A high-level depiction of Haley's architecture.

We note that our focus is on automatically generating the control flow of the composition and representing it using BPEL. If inputs for WSs participating in the compositions are available either from outputs of previous invocations or *a priori*, the resulting BPEL is executable. As we discuss later, we do not address the problem of data mediation, which is a separate and challenging research problem in itself.

[1]Haley is available for download at **http://denali.cs.uga.edu/haley**

1.3 RESTful Web Service Composition

By applying the principles of REST in Web service development, RESTful WSs [81] are emerging as the choice for many of the leading Internet companies to expose their internal data and functionalities as URI identified resources. Some of the advantages of RESTful WSs include:

Light-weight: RESTful WSs directly utilize HTTP as the invocation protocol which avoids unnecessary XML markups or extra encapsulation for APIs and input/output. The response is the representation of the resource itself, and does not involve any extra encapsulation or envelopes. As a result, RESTful WSs are easier to develop and consume than WSDL/SOAP WSs, especially as Web Application Programming Interface (API) [8] in the Web 2.0 [69, 70] context. Additionally, they depend less on vendor software and mechanisms that implement the additional SOAP layer on top of HTTP. RESTful WSs usually deliver better performance due to the light-weight nature.

Easy-accessibility: URIs used for identifying RESTful WSs can be shared and passed around to any dedicated service clients or common purpose applications for reuse. The URIs and the representation of resources are descriptive and thus makes RESTful WSs easily accessible [92]. RESTful WSs have been widely used to build Web 2.0 applications and mashups.

Scalability: The scalability of RESTful WSs comes from its ability to naturally support caching and parallelization/partitioning on URIs. The responses of GET (a side-effect free operation) can be cached exactly the same as web pages are currently cached in the proxies, gateways and content delivery networks (CDNs). Additionally, RESTful WSs provide a simple and effective way to support load balancing based on URI partitioning. Compared to ad-hoc partitioning of functionalities behind the SOAP interfaces, URI-based partitioning is more generic and flexible, and could be easier to realize.

Declarative: In contrast to imperative services from the perspective of operations, RESTful WSs take a declarative approach and view the applications from the perspective

of resources. Being declarative means that RESTful WSs focus on the description of the resources themselves, rather than describing what the functions are performed. Declarative style brings the fundamental differences between RESTful WSs and WSDL/SOAP WSs to the forefront. While building services for a particular system, the declarative approach focuses on what resources need to be exposed and how these resources can be represented; imperative approach focuses on what operations need to be provided and what are the input/ouput of these operations. Declarative approach is considered to be a better choice [97] to build flexible, scalable and loosely-coupled SOA systems.

The characteristics of RESTful WSs mentioned above make automated RESTful Web service composition a *fundamentally different* problem than the composition problem of WSDL/SOAP WSs. Although the research community has put significant effort on automating WSDL/SOAP WSs, automated RESTful Web service composition problem, to the best of our knowledge, is less explored.

In Chapter 6, we outline the challenges of this particular problem and present our initial effort towards the problem of automating RESTful Web service composition. Our main contribution in this regard is the introduction of a formal description of the RESTful Web service composition problem [106]. While analyzing the differences and challenges involved in this problem, we propose a formal model for classifying and describing RESTful WSs, and present a situation calculus [52, 80] based state transition system for composing them automatically. More details will be presented in Chaper 6.

1.4 Contributions

Facilitating the assembly of services to form composite services is an important functionality in SOA. Users may combine and reuse WSs toward the production of new applications. In this dissertation, we focus on the problem of automatically assembling WSs to form compositions with emphasis on modeling WS uncertainty, optimizing the process and improving the scalability. Two related challenges involved in automated Web service composition problem

are the automatic construction of the service flow and mediating the data heterogeneity. Our approach focuses on automatically constructing the control flow based on the functional and non-functional requirements of the composition. It may be extended using a data mediation module capable of handling data heterogeneity issues.

Generally speaking, our research focuses on WSDL/SOAP WS composition and RESTful WS composition. We have performed an extensive study of the existing Web service composition research and identified some of the key issues needed to be solved by WSC approaches. We have proposed a novel hierarchical, symbolic decision-theoretic planning based framework and empirically evaluated the framework through its applications. Our approach investigates some of the issues that have not been addressed before and demonstrates many unique advantages over prevalent approaches. In this section, we summarize our contributions in the dissertation, and present more detailed description in the later chapters of this dissertation.

1.4.1 Hierarchical SMDP Based Approach

We adopt decision-theoretic planning for composing WSs into processes. Decision-theoretic planners such as Markov decision processes (MDPs) [13] generalize classical planning techniques to nondeterministic environments where action outcomes may be uncertain, and associate costs to the different plans thereby allowing the selection of an optimal plan. Previously, Doshi et al. [31] showed how we may use MDPs to compose WS-based workflows. Compared to classical AI planners, a decision-theoretic planner probabilistically models the uncertainty inherent in WSs and facilitates a cost-based process optimization. Being stochastic, MDPs bring in the ability to model uncertain behaviors of WSs with non-deterministic actions, and they associate costs (or rewards) to the different outcomes of actions and resulting states. These techniques are especially relevant in the context of SOAs where services may fail and processes must minimize costs.

While the previous work [31] demonstrated that decision-theoretic planning addresses both the uncertainty and optimality issues, the scalability issue remains unresolved. To deal

with this issue, we examine the application of hierarchical semi-MDP planning toward the hierarchical composition of WSs [104]. We present a hierarchical approach for composing processes that may be nested – some of the components of the process may be sub-processes themselves. The major contributions of this work are listed below:

Contributions

- In order to represent the invocation of lower level processes whose execution times are uncertain and different from simple service invocations, we model WS invocations using actions in Semi-Markov decision process (SMDP) [74] that generalizes MDPs by allowing actions to be temporally extended.

- Many real world processes are amenable to a hierarchical decomposition into lower level processes and primitive service invocations. We present a new hierarchical framework for modeling, composing, and executing large scale processes by exploiting such a hierarchy. We model the compositions with only primitive WSs as primitive SMDPs and the ones with sub-processes as composite SMDPs.

- A method of constructing the primitive SMDP planning problem from the Web service composition problem is proposed, and more importantly we provide ways for deriving the parameters of the composite SMDPs from the lower level ones.

1.4.2 Haley: Hierarchical First-order SMDP Based Approach

Although hierarchical SMDP based approach promotes scalability, along with existing WSC techniques, it is further plagued by two challenges:

(i) As the number of WSs increases, there is an explosion in the size of the state space;

(ii) There is a growing consensus among the WS description standards such as OWL-S [50] and SAWSDL on using first order logic (or its variants) to logically represent the preconditions and effects of WSs.

However, many of the existing planning techniques used for WS composition do not use the full generality of first order logic while planning. We propose a first-order hierarchical decision-theoretic planning framework, which we call Haley [105], for WSC problems. Haley improves on previous work by allowing WS composition at the logical level. Specifically, Haley enables composition using the first order sentences that represent the preconditions and effects of the component WSs. In addition to using a symbolic representation, Haley promotes scalability by exploiting the hierarchical decomposition of real-world processes. Similar to the idea of hierarchical decomposition in hierarchical SMDP based approach, we extend it to the hierarchical decomposition of first-order SMDP (FO-SMDPs).

Contributions

- Haley offers a way to mitigate the problem of exponentially growing state spaces by composing at the logic level and preserves the expressiveness of first order logic. Haley operates directly on first order logic based representations of the state space to obtain the compositions. As a result, it supports an automated elicitation of the corresponding planning domain from Web service descriptions and produces a much more compact domain representation than classical AI planners.

- Haley models each level of the hierarchy in the process using a FO-SMDP that extends a SMDP to operate directly on first order logic sentences, which provide a logical representation of the traditional state space. In particular, the lowest levels of the hierarchy (leaves) are modeled using a FO-SMDP containing *primitive* actions which are invocations of the WSs. Higher levels of the process are modeled using FO-SMDPs that contain *abstract* actions, which represent the execution of lower level processes. We represent their invocations as *temporally extended* actions in the higher level FO-SMDPs.

- Since descriptions of only the individual WSs are usually available, we provide methods for deriving the model parameters of the higher level FO-SMDP from the parameters

of the lower level ones. Thus, our approach is applicable to WSCs that are nested to an arbitrary depth.

- Our experiments demonstrate the advantages of a hierarchical decomposition and logic based representation. The hierarchical approach consumes less planning time than the flat approach; and the first order representation produces more compact planning domains.

1.4.3 An End-to-End WS Composition Tool Suite

Due to the limitations of the existing approaches mentioned previously and the complexity of the WS composition problem itself, few implemented tools exist, although many approaches have been proposed in the literature. We have implemented Haley[2] and provide a comprehensive tool suite. The suite accepts WSs described using standard languages such as SAWSDL. It provides process designers with an intuitive interface to specify process requirements, goals and a hierarchical decomposition, and automatically generates executable BPEL processes, while hiding the complexity of planning and BPEL from users.

Contributions

- We have designed a general first-order SMDP based decision-theoretic planner, eDT-GOLOG, which may be used independently of Haley. eDT-GOLOG is used as the planning engine in Haley to generate the plan based on WS descriptions and user-specified goals.

- To help process designers view the functionalities of individual WSs, we designed a SAWSDL viewer to visualize the IOPE –Input, Output, Preconditions and Effects – of WSs. To the best of our knowledge, this is the first graphical viewer for SAWSDL described WSs.

[2]Haley is available for download at **http://denali.cs.uga.edu/haley**

- A process hierarchy modeler to help process designers specify the hierarchy in the processes. Process designers may utilize this tool to form hierarchies by grouping WSs in an intuitive way.

- An integrated environment for the process designers to (1) import candidate WSs and their description files, (2) specify process hierarchies, initial state and goals, (3) generate the planning domain, (4) generate the plan and (5) convert the plan into the corresponding BPEL file.

1.4.4 An Automated RESTful Web Service Composition approach

Automated RESTful WS composition is much less studied in the WS research community than WSDL/SOAP WS composition. We have put our initial efforts into RESTful WS modeling and composition. We are hoping our research will attract more interests and engage more researches from the WS research community.

Contributions

- As an early research effort towards automated RESTful WS composition, we introduce and formally define this problem. In addition, we analyze the differences between automated WSDL/SOAP WS composition and automated RESTful WS composition. We identify a set of challenges for addressing automated RESTful WS composition.

- To facilitate automated RESTful WS composition, we classify and formally model RESTful WSs. A situation calculus based state transition system has been studied to automate the composition of RESTful WSs.

1.5 Structure of the Dissertation

This dissertation covers two closely connected topics: automated WSDL/SOAP WS composition and RESTful WS composition. While sharing a similar objective and some common

properties in the context of SOA application, these two composition problems are fundamentally different due to the two different perspectives of building WSs. With this in our mind, this document is structured so that we present in the early chapters our proposed framework solving the automated WSDL/SOAP WS composition problem, and introduce our effort towards the automated RESTful WS composition problem. Challenges and solutions to these two composition problems have been discussed and compared in the later chapters.

We briefly summarize the structure of the dissertation below:

We describe two motivating scenarios in Chapter 2, both of which exhibit nested structure with 2 levels and 3 levels respectively. The second scenario will be used as a running example to illustrate our approach to automated WSDL/SOAP WS composition.

In Chapter 3, we introduce the background knowledge of Markov Decision Processes (MDPs), Semi-Markov Decision Processes (SMDPs) and First Order MDPs (FO-MDPs). In addition, we discuses probabilistic situation calculus using specific examples. Probabilistic situation calculus is the representation language used in FO-SMDPs.

We present in Chapter 4 the theoretical framework of Haley. We introduce First Order Semi-Markov Decision Processes (FO-SMDPs) and explain a hierarchical FO-SMDP based decision-theoretic planning framework. This planning framework is used in Haley as the planning engine to automatically compose WSs into processes. We will detail how Haley approaches an order handling scenario (Fig. 2.2) in this chapter.

In Chapter 5, we present the system modules and implementation details. We also discuss the generation and the execution algorithms of processes. Experiment results will be presented and analyzed in detail.

Detailed description of our approach to automated RESTful WS composition is presented in Chapter 6. We introduce a classification based formal model for describing RESTful WSs. A state transition system approach is proposed to automate the composition of RESTful WSs.

In Chapter 7, we survey existing planning based approaches and configuration based approaches to WS composition and briefly compare them with Haley. In addition, we cover the related proposals for modeling RESTful WSs. Mashup is also mentioned and compared to RESTful WS composition.

Finally, we conclude our work in Chapter 8 with a summary of original contributions and present some future research directions.

Chapter 2

Motivating Scenarios

In order to illustrate our framework, we briefly describe two motivating scenarios that exhibit hierarchical structures. We intend to form WSCs that meet the goal using the specified input, output, precondition and effect (IOPE) for each WS. Furthermore, in the first scenario, we will optimize over the traditional QoS parameters advertised for the WSs (interface parameters) such as response time, invocation cost and reliability. However, the structure of a WSC may not depend on the interface parameters alone. Hence, in the second scenario, we will additionally optimize over domain parameters such as the cost of the service and availability of the products.

2.1 Online Shopping

In the first scenario, we study a typical online shopping process (Fig. 2.1). A Chinese college student would like to know the total cost of buying a textbook from the US portal of Amazon because the desired textbook is not published in China. The total cost includes the book's price and the price of shipping it to China, both of which are, of course, in US dollars. A currency conversion WS is utilized to convert US dollars into Chinese currency, Yuan. Two of the components, *GetBookPriceInYuan* to get the price of the book and, *GetShipping-CostInYuan* to get the shipping cost, are actually subprocesses composed of primitive WSs. To make this scenario realistic, we obtain these WSs from actual Web application sites such as the USPS web tools [5], Amazon WSs [1] and WebserviceX.net [6].

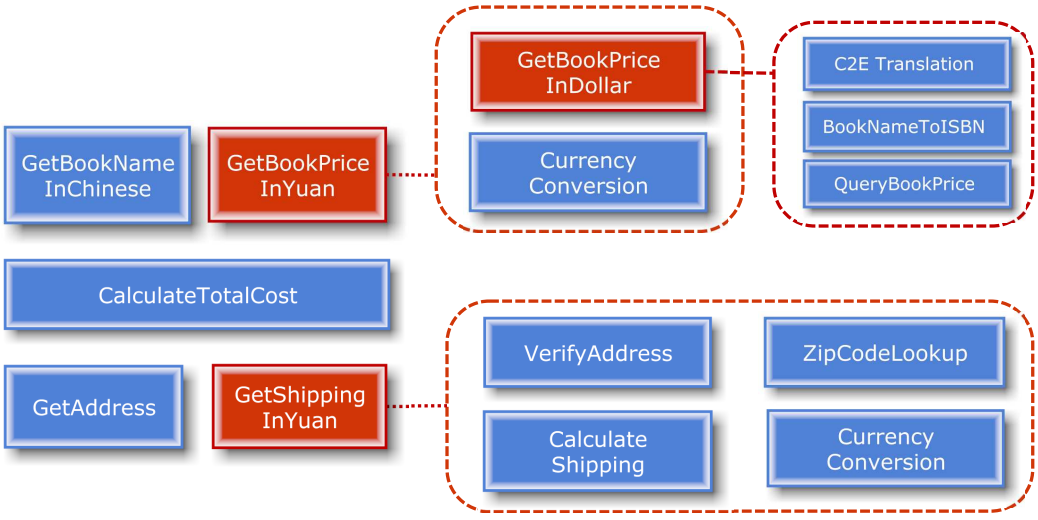

Figure 2.1: A 3-level hierarchical online shopping scenario in which the service *GetBookPriceInYuan* and *GetShippingCostInYuan* are subprocesses. *GetBookPriceInDollar* in subprocess *GetBookPriceinYuan* is also a subprocess composed of WSs.

2.2 Order Handling Scenario in Supply Chain

Our second example is typical of scenarios for handling orders that are parts of the supply chains of manufacturers (Fig. 2.2). This scenario will be used as a running example to illustrate our approach throughout the document.

An instance of the business process is created when a customer sends in an order. The order specifics first need to be verified, in that the customer needs to be checked and her payment needs to be processed. Subsequently, the manufacturer checks for supplies that are required to complete the order. In this step, he may choose to check his own inventory first and then ask his preferred supplier, if his own inventory is deficient. Alternately, he may elect to directly ask his preferred supplier for goods, since he does not expect his inventory to satisfy the order. A final resort is the spot market which is guaranteed to fulfill his order. On receiving the supplies, the manufacturer will ship the completed order to the customer.

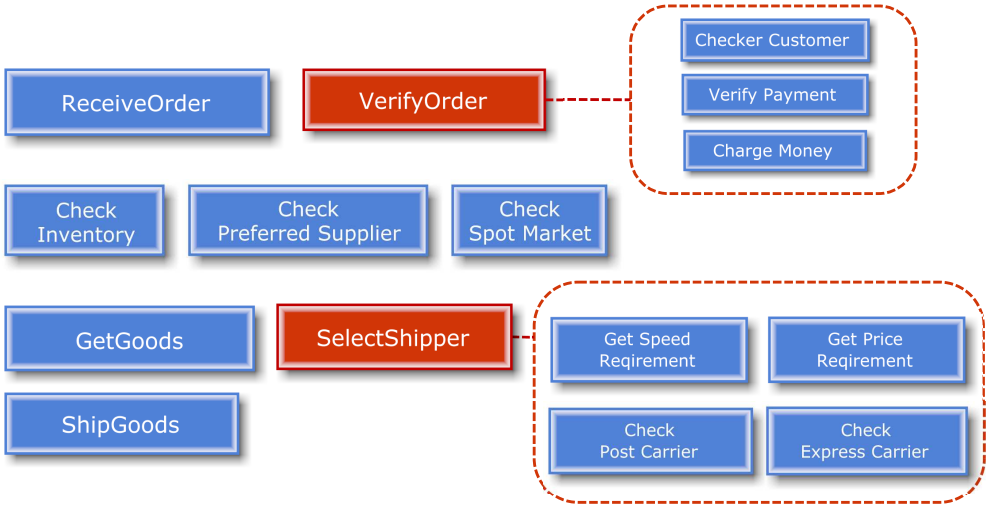

Figure 2.2: A 2-level hierarchical order handling scenario as a part of the supply chains of manufacturers in which the service *Verify Order* and *Select Shipper* are subprocesses themselves. The three suppliers (Inventory, Preferred Supplier and Spot Market) in this scenario have different costs and probabilities of order satisfaction.

Note that the three candidate suppliers differ in their probabilities and costs of order satisfactions. In particular, while the inventory exhibits a low cost of satisfying the order, the spot market is the most expensive among the three. However, it is also guaranteed to satisfy the order, while the inventory has the least probability of doing so. This particular scenario illustrates that not only are the WS interface parameters (IOPE and QoS) important factors while composing WSC, but other domain-specific service parameters such as order cost and the rate of satisfying orders also need to be considered while determining the composition.

In this case, we may combine the WS invocation cost and the order cost to produce the total cost of using the WSs. The availability of each service is determined by two probabilities: the WS interface reliability and the probability of order satisfaction. For example, the probability that the *Preferred Supplier* is available to satisfy the order is a product of the probability that the preferred supplier's WS is working properly and the probability that

this order can be satisfied by it. In other words, order satisfaction is contingent on both, the WS is working properly and the preferred supplier has sufficient products in stock.

Chapter 3

Background

In this section, we briefly describe Markov decision processes (MDPs) [74] and semi-Markov decision processes (SMDP) [74], a temporal generalization of MDP. MDPs and SMDPs are well-known framework for decision-theoretic planning [15, 17]. And then we focus on first-order extension of MDPs (FO-MDPs) that allow planning on first order logic representations expressed with situation calculus.

3.1 Markov Decision Processes

MDPs [74] model the process environment as a tuple,

$$\mathrm{M}DP = \langle S, A, T, R, s_0 \rangle \qquad\qquad where$$

- S is a set of states with each state often factored into variables
- A is a set of actions
- $T : S \times A \rightarrow \Delta(S)$, is the transition function representing the uncertain effects of actions where $\Delta(\cdot)$ is the set of probability distributions
- $R : S \times A \rightarrow \mathbb{C}$ is the reward function representing the reward/cost of performing actions
- $s_0 \in S$ is the start state

Solution of an MDP results in a policy, which is a mapping from states to actions. In order to solve the MDP, we associate with each state a value, $V^n(s)$, that represents the expected cost of performing an optimal sequence of actions from that state. We define this value using the following equation, which forms the basis for *value iteration* [13]:

$$V^n(s) = \underset{a \in A}{max}\ R(s,a) + \sum_{s' \in S} T(s'|s,a) V^{n-1}(s') \tag{3.1}$$

Standard solution technique involves iterating over Eq. 3.1 until $V^n(s)$ converges. The converged value is the maximal utility for each state, and the corresponding mapping from states to actions that maximizes utilities is the optimal policy. The optimal action(s) to perform from a state is then the one which results in the lowest expected cost. An application of MDPs to WSC is given in [31].

3.2 Semi-Markov Decision Processes

A semi-Markov decision process (SMDP) [74] is a temporal generalizations of MDP. Instead of assuming that the durations of all actions are identical and therefore ignoring them while planning, SMDPs explicitly model the system evolution in continuous time and model the time spent in a particular state while performing an action as following a pre-specified probability distribution. Analogous to an MDP, solution to a SMDP produces a *policy*. The policy assigns to each state of the WSC action(s) that is expected to be optimal over the period of consideration. We formally define a SMDP that models the composition problem as a tuple:

A SMDP is defined as a tuple,

$$SMDP = \langle S, A, T, R, K, F, C, s_0 \rangle$$

- $S = \Pi_{i=1}^{n} X^i$, where S is the set of all possible states factored into a set, X, of n variables, $X = \{X^1, X^2, \ldots, X^n\}$
- A is the set of all possible actions

- T is the transition function, $T : S \times A \rightarrow \Delta(S)$, where $\Delta(\cdot)$ specifies a probability distribution. The transition function captures the uncertain effect of performing an action on particular variables
- K is the lump sum reward, $K : A \rightarrow \mathbb{R}$. This specifies the reward (or cost) obtained on performing an action
- F is the sojourn time distribution for each action, $F : A \rightarrow \Delta(t)$, where $t \in [0, T_{max}]$, T_{max} is the maximum time duration of any action. Given the action, a, the system will remain in the state for a certain amount of time, t, which follows a density described by $f(t|a)$. Note that the sojourn time distribution may also depend on the current state. This distribution represents the varying response times of WS invocations;
- C is the cost accumulating rate, $C : A \rightarrow \mathbb{R}$, which specifies the rate at which the cost accumulates on performing a temporally extended action.
- $s_0 \in S$ is the start state of the process

We may describe an example evolution of a SMDP as follows: At time t_0, the system occupies state s_0, and the WSC chooses action a_0 based on a particular policy. Consequently, the system remains in s_0 for t_1 time units after which the system state changes to s_1, and the next decision epoch occurs. The WSC performs action a_2, and analogous sequences of events follow. The sequence $\{t_0, s_0, a_0, t_1, s_1, a_1, \ldots, t_n, s_n\}$ denotes the history of SMDP up to the n^{th} decision epoch. $\{t_0, t_1, t_2, \ldots, t_n\}$ are the sojourn times between two consecutive decision epochs. While in a MDP, these times are fixed, in a SMDP, the time durations follow certain probability distributions given by F.

In order to solve the SMDP, we define the following:

$$R(s, a) = RS(s) - (K(a) + C(a) \int_0^{T_{max}} e^{-\alpha t} f(t|a) dt) \tag{3.2}$$

Notice that we subtract the expected cost of performing the action, a, from the reward obtained at the state, s. We model the total expected cost using two parts – state dependent

reward RS(S) and action dependent costs. $f(t|a)$ is the probability density function of the sojourn time distribution.

Analogous to MDPs, we associate a value function, $V : S \rightarrow \mathbb{R}$, with each state. This function quantifies the desirability of a state over the long term.

$$V^n(s) = \max_{a \in A} R(s,a) + \sum_{s' \in S} M(s'|s,a) V^{n-1}(s') \tag{3.3}$$

where:

$$M(s'|s,a) = \int_0^{T_{max}} e^{-\alpha t} T(s'|s,a) f(t|a) dt \tag{3.4}$$

Standard SMDP solution techniques [13] for arriving at the optimal policy involve repeatedly iterating over Eq. 3.3 until the function, V, approximately converges. Another technique for computing the policy requires formulating and solving a linear program (LP).

Traditionally, the state in MDPs and SMDPs is represented using propositions and a combination of all possible values of the propositions becomes the state space. Notice that the size is exponential in the number of propositions. In order to solve MDPs or SMDPs, we must explicitly enumerate over all pairs of states and actions. This becomes a computational challenge when composing large processes.

In addition, WS description standards such as OWL-S and SAWSDL seek to express the preconditions and effects of WSs using first order logic based languages such as RuleML [39]. The traditional MDP based approach [31] composes these WSs by grounding and propositionalizing the WS descriptions. It fails to scale to large WS composition problems because the size of the state space grows exponentially with the number of objects.

Efforts have been proposed to logically represent MDPs and solve them symbolically to avoid an explicit enumeration of the states. These approaches include symbolic dynamic programming [18] using situation calculus, fluent calculus based value iteration [40] and a relational bellman algorithm (Rebell) [43]. Automated WS composition needs for a decision-theoretic planning framework that operates symbolically on first order logic descriptions. In [18], Boutilier et al. introduced first order MDPs (FO-MDPs) that use a probabilistic

variant of first order situation calculus to logically represent the domain and use symbolic value iteration to solve the FO-MDP. We cover the details about FO-MDP in Section 3.4.

Before we describe FO-MDPs, we introduce situation calculus using the supply chain example(See Figure 2.2). Probabilistic Situation Calculus is the representation language used in FO-MDPs.

3.3 Probabilistic Situation Calculus

Situation calculus [52, 79] is a first order logic based framework for representing dynamic environments and actions, and reasoning about them. It uses *situations* to represent the state of the world, and *fluents* to describe the changes from one situation to the other caused by the actions. We briefly explain the components of the probabilistic variant of situation calculus:

• **Actions** are first order terms, $A(\vec{x})$, each consisting of an action name, A, and its argument(s), $\vec{x}$. For example, the action of receiving an order from the customer can be denoted as $ReceiveOrder(o)$; the action of checking inventory can be denoted as $CheckInventory(o)$, where o is a variable denoting the order.

• A **situation** is a sequence of actions describing the state of the world and usually represented by symbol $do(a, s)$. For example, $do(ReceiveOrder(o), s_0)$ denotes the situation obtained after performing $ReceiveOrder(o)$ in the initial situation s_0 (s_0 is a special situation which is not represented using a do function). Situation $do(ChargeMoney(o), do(VerifyPayment(o), do(CheckCustomer(o), s_0)))$ represents the situation obtained after performing the action sequence ($CheckCustomer(o)$, $VerifyPayment(o)$, $ChargeMoney(o)$) in s_0.

• **Fluents** are situation-dependent relations and functions whose truth values vary from one situation to another. For example, $HaveOrder(o, s)$ means the process has received an order denoted by o in situation s; $ValidCustomer(o, s)$ means the customer identified in o has been validated in s.

• **Nature's choices:** Situation calculus in its original form is restricted to deterministic actions, while MDPs allow stochastic actions and are designed to make decisions under uncertainty. To model stochastic actions in situation calculus, we decompose a stochastic action, $A(\vec{x})$, into a set of deterministic actions, $n_j(\vec{x})$, each of which is selected randomly. We assume that this choice may be made by nature. For example, an action invoking a WS that has multiple effects could be decomposed into deterministic actions for each effect. Nature's choices, introduced in [18], support stochastic actions in situation calculus. A stochastic action is decomposed into deterministic actions under Nature's choice - it chooses the deterministic action with some specified probability, that actually gets executed when an agent performs a stochastic action. We give nature's choices intuitive meanings in our framework. As we mentioned previously, we assign a probability to each possible WS invocation outcome including the positive outcome and negative outcomes like service failure. For example:

$choice(CheckCustomer(o), a) \equiv a = CheckCustomerS(o) \vee a = CheckCustomerF(o)$

$choice(CheckPreferredSupplier(o), a) \equiv a = CheckPreferredSupplierS(o) \vee$

$a = CheckPreferredSupplierF(o)$

where $CheckCustomerS(o)$ and $CheckCustomerF(o)$ denote the deterministic actions that succeed (customer validated) or fail, respectively.

• **Probabilities for nature's choices:** For each possible outcome, $n_j(\vec{x})$, associated with stochastic action, $A(\vec{x})$, we specify the probability, $Pr(n_j(\vec{x}), A(\vec{x}), s)$ with which the outcome will occur, given that $A(\vec{x})$ was performed in situation s. For example:

$Pr(CheckCustomerS(o), CheckCustomer(o), s) = 0.9$

$Pr(CheckCustomerF(o), CheckCustomer(o), s) = 0.1$

$Pr(CheckPreferredSupplierS(o), CheckPreferredSupplier(o), s) = 0.72$

$Pr(CheckPreferredSupplierF(o), CheckPreferredSupplier(o), s) = 0.28$

To calculate $Pr(CheckPreferredSupplierS(o), CheckPreferredSupplier(o), s)$, we need to take into account both the service availability and the probability of order satisfaction. If the service availability of Preferred Supplier WS is 0.9, and its probability of order satisfaction is 0.8, the overall probability that Preferred Supplier WS returns YES is 0.72.

While four classes of axioms are used in situation calculus to axiomatize a domain, we additionally focus on the precondition and successor state axioms that are important for FOMDPs.

• **Action precondition axioms:** We define one axiom for each action: $Poss(a(\vec{x}), s) \equiv \Pi(\vec{x}, s)$, which characterizes the precondition of the action. For example, the precondition axiom of action $CheckCustomer(o)$ is:

$$HaveOrder(o, s) \Rightarrow Poss(CheckCustomer(o), s)$$

where $Poss$ denotes the possibility of performing the action.

• **Successor state axioms(SSA)** are axioms that describe the effects of actions on fluents. Hence there is one such axiom for each fluent. Successor state axioms provide a way to address the frame problem – the problem of representing all the things that stay the same on performing an action. Action effects are compiled into successor state axioms, where the truth value of a fluent is completely determined by the current situation s and the action to

be performed. There is one such axiom for each fluent:

$$F(\vec{x}, s)\text{: } F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s).\ \Phi_F(\vec{x}, a, s) = \gamma^+(\vec{x}, a, s) \vee (F(\vec{x}, a) \wedge \neg\gamma^-(\vec{x}, a, s))$$

where $\gamma^{+/-}(\vec{x}, a, s)$ contains all the combinations of actions and conditions that would make fluent F true/false respectively. For instance of our example domain, the successor state axiom for $ReceiveOrder(o)$:

$$Poss(a, s) \Rightarrow HaveOrder(o, do(a, s)) \Leftrightarrow a = ReceiveOrderS(o) \vee (HaveOrder(a, s) \wedge a \neq CancelOrderS(o))$$

In other words, we have the order in the situation that results from performing the action if and only if we performed the $ReceiveOrderS(o)$ action or we already have it in the current situation and do not perform an action that will cancel the order. We give two more example SSAs in the example domain for Fluent $ValidCustomer(o)$ and $ValidPay(o)$:

$$Poss(a, s) \Rightarrow ValidCustomer(o, do(a, s)) \Leftrightarrow a = CheckCustomer(o) \vee (ValidCustomer(o, s)$$
$$Poss(a, s) \Rightarrow ValidPay(o, do(a, s)) \Leftrightarrow a = VerifyPayment(order) \vee (validPay(order, s)$$

• **Regression:** Regression is a mechanism for proving consequences in situation calculus. It is based on expressing a sentence containing the situation $do(a, s)$ in terms of a sentence containing the action a and the situation s, without the situation $do(a, s)$. The regression of a sentence φ through an action a is φ' that holds prior to a being performed iff φ holds after a. Successor state axioms support regression in a natural way [18]. Suppose that a fluent F's successor state axiom is $F(\vec{x}, do(a, s)) \Leftrightarrow \Phi_F(\vec{x}, a, s)$, we inductively define the regression of a sentence whose situation arguments all have the form $do(a, s)$:

$Regr(F(\vec{x}, do(a, s))) = \phi_F(\vec{x}, a, s)$

$Regr(F(\neg\psi) = \neg Regr(\psi)$

$Regr(F(\psi_1 \wedge \psi_2) = Regr(\psi_1) \wedge Regr(\psi_1)$

$Regr(F(\exists x\psi) = (\exists x)Regr(\psi)$

3.4 First Order Markov Decision Processes

We briefly present First Order Markov Decision Processes (FO-MDP) formalism and the symbolic dynamic programming solution using the supply chain example. We refer the reader to [18, 83] for more details.

Actions in FO-MDPs are the stochastic actions decomposed into nature's choices. To simply the presentation, FO-MDPs introduce *case notation* to represent the transition and cost functions. A case notation is defined as, $case[\phi_1(s), t_1; \ldots; \phi_n(s), t_n]$ where ϕ_i, $i = 1 \ldots n$ represents a first order logic sentence in situation calculus, t_i is the corresponding term. We note that the case notation *partitions* the state space into n classes; within a class i each state unifies with $\phi_i(s)$ (ie., $\phi_i(s)$ is true for state s).

Let $A(\vec{x})$ be a stochastic action with choices, $n_1(\vec{x})$, $\ldots$, $n_k(\vec{x})$, then the choice probabilities are specified as: $pCase(n_j(\vec{x}), A(\vec{x}), s) = case[\phi_1(s), p_1^j; \ldots; \phi_n(s), p_n^j]$. Note that we will have one such $pCase$ for each j. For example,

$pCase(CheckCustomerS(o), CheckCustomer(o), s) = case[true, 0.9]$

$pCase(CheckCustomerF(o), CheckCustomer(o), s) = case[true, 0.1]$

The reward function may be represented in case notation:

$rCase(s) = case[\xi_1(s), r_1; \ldots; \xi_n(s), r_n)]$. For example,

$rCase(s) = case[ValidCus(o) \wedge ValidPay(o) \wedge Charged(o), 10; \neg(ValidCus(o) \wedge ValidPay(o) \wedge$

$Charged(o)), 0]$

Intuitively, an operation on two case notations takes the cross-product of their cases (partitions) and performs the corresponding operation on the terms of the paired partition. The following operations are defined in case notations:

$$case[\phi_i, t_i : i \leq n] \oplus case[\psi_j, t_j : j \leq m] = case[\phi_i \wedge \psi_j, t_i + t_j : i \leq n, j \leq m]$$
$$case[\phi_i, t_i : i \leq n] \ominus case[\psi_j, t_j : j \leq m] = case[\phi_i \wedge \psi_j, t_i - t_j : i \leq n, j \leq m]$$
$$case[\phi_i, t_i : i \leq n] \otimes case[\psi_j, t_j : j \leq m] = case[\phi_i \wedge \psi_j, t_i \cdot t_j : i \leq n, j \leq m]$$

We illustrate how cases partition the state space and an operation on case notations in Fig. 3.1.

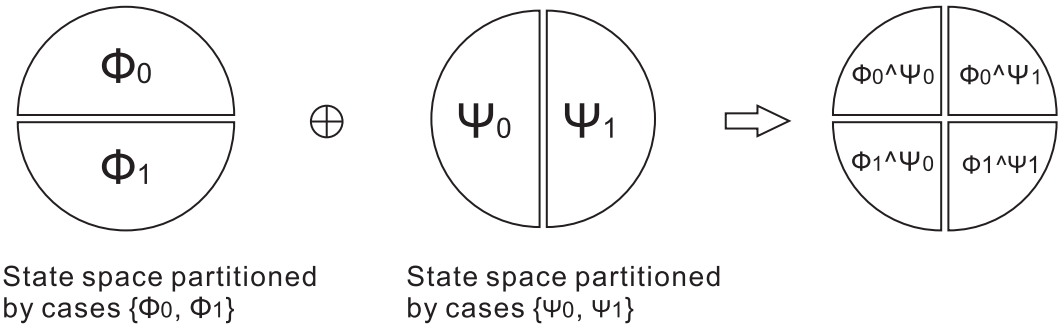

Figure 3.1: Cases partition the state space. Within a class i, each state unifies with $\phi_i(s)$ (ie., $\phi_i(s)$ is true for state s). Here, one case notation partitions the state space with cases ϕ_0 and ϕ_1 and the other one partitions the state space with cases ψ_0 and ψ_1. The operation on the two case notations takes the cross-product of their cases.

On representing the FO-MDP parameters in case notation and axiomatizing action preconditions and effects, we may perform symbolic value iteration to solve the FO-MDP. We briefly introduce the idea here and refer the readers to [18, 83] for more details. We first define first order decision-theoretic regression (FODTR) as:

$$FODTR(vCase(s), A(\vec{x})) = \gamma \cdot [\oplus_j \; pCase(n_j(\vec{x}), s) \otimes Regr(vCase(do(n_j(\vec{x}), s)))]$$

Here, $vCase$ is the case notation for the value function, V^n.

$$Regr(vCase(s), A(\vec{x})) = rCase(s) \oplus FODTR(vCase(s), A(\vec{x})) \tag{3.5}$$

where $Regr$ on the left regresses the value function through action, $A(\vec{x})$, and produces a case notation with action parameters as free variables. The parameters may be removed from consideration through existential quantification: $Regr(vCase(s), A) = \exists \vec{x}\ Regr(vCase(s), A(\vec{x}))$ The optimal value, analogous to Eq. 3.1, is given by the action that maximizes the action-value pair:

$$Regr(vCase(s)) = max_A\ Regr(vCase(s), A)$$

Chapter 4

Haley: A Hierarchical Framework for Logical Composition of Web Services

Many real world business processes are amenable to a hierarchical decomposition into lower level processes and primitive service invocations. We present a new framework, which we call Haley, for modeling, composing, and executing large WSCs by exploiting such a hierarchy. Our approach is to use first order *semi*-Markov decision processes (FO-SMDPs), which are temporal generalizations of the FO-MDPs mentioned in Section 3.4 to perform the composition. Specifically, they allow temporally extended actions of uncertain durations, which we call *abstract actions*. The actions are used to represent the invocations of lower level WSCs.

Haley models the lowest level service composition problem using primitive FO-SMDPs, while higher level compositions are modeled using composite FO-SMDPs (Fig. 4.1). We also show how the model parameters of the composite FO-SMDPs in the case of abstract actions may be derived from the parameters of the primitive FO-SMDPs that signify the actions. Parameters of the primitive FO-SMDPs are, of course, obtained directly from the relevant WS descriptions. To the best of our knowledge, Haley is the first framework that combines hierarchical decomposition with logic (knowledge) level composition of WSs, thereby offering a scalable approach capable of operating directly on first order logic based descriptions of WSs.

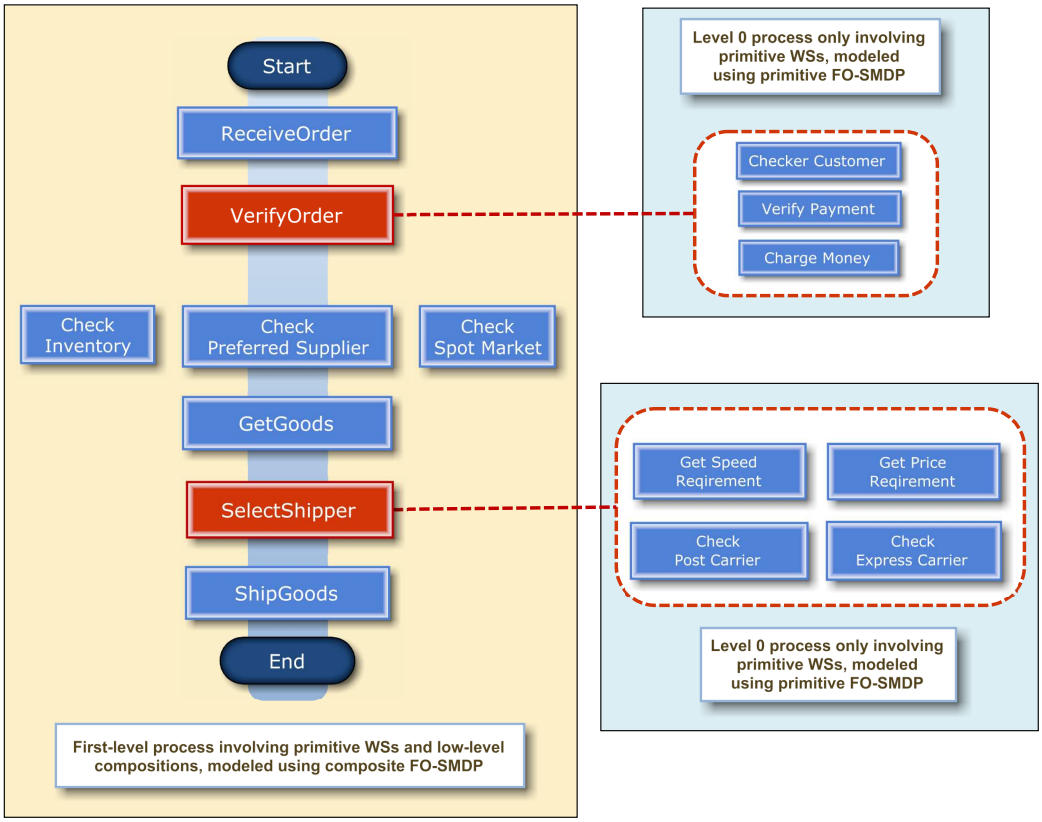

Figure 4.1: High-level process in the order handling scenario with low-level sub-processes is composed using composite FO-SMDP. Low-level processes with only primitive WSs are composed using primitive FO-SMDP.

4.1 First Order Semi-Markov Decision Processes

Standard solution of the SMDP involves repeatedly iterating over Eq. 3.3 for the desired number of steps or until the function, V, approximately converges. The optimal policy from each state is then the action which results in the maximum value of that state.

We now extend the classical SMDPs described in Section 3.2 to first order SMDPs, in a manner similar to Section 3.4. This not only avoids an enumeration over all state-action pairs but also allows us to operate directly on first order logic based descriptions of WS preconditions and effects.

Analogous to FO-MDPs, we adopt the probabilistic situation calculus to logically represent the FO-SMDP. The parameters S, A, T, and R of the FO-SMDP are as defined in Section 3.4 using case notation. We give the case notations for the new parameters specific to SMDPs next.

The lump sum cost function, $K(A(\vec{x}))$, may be represented in case notation as:

$$kCase(A(\vec{x})) = case[\beta_1(A(\vec{x})), k_1; \ldots; \beta_{|A|}(A(\vec{x})), k_{|A|}]$$

where $|A|$ is the number of actions. Similarly the accumulating rate, $C(A(\vec{x}))$, represented in case notation is:

$$cCase(A(\vec{x})) = case[\beta_1(A(\vec{x})), c_1; \ldots; \beta_{|A|}(A(\vec{x})), c_{|A|}]$$

The sojourn time distribution, $F(A(\vec{x}))$, in case notation is:

$$fCase(A(\vec{x})) = case[\beta_1(A(\vec{x})), f_1(t); \ldots; \beta_{|A|}(A(\vec{x})), f_{|A|}(t)]$$

Here $\beta_i(A(\vec{x}))$ is defined as, $\beta_i(A(\vec{x})) : \ A(\vec{x}) = a_i \ \ i = 1, 2, \ldots, |A|$. Intuitively, K, C, and F have different values or functions for each action. For example,

$kCase(A(\vec{x})) = case[A(\vec{x}) = CheckCustomer(o),\ 2;\ A(\vec{x}) = VerifyPayment(o),\ 3;\ A(\vec{x}) = ChargeMoney(o),\ 2]$

$cCase(A(\vec{x})) = case[A(\vec{x}) = CheckCustomer(o),\ 0.2;\ A(\vec{x}) = VerifyPayment(o),\ 0.2;\ A(\vec{x}) = ChargeMoney(o),\ 0.2]$

$fCase(A(\vec{x})) = case[A(\vec{x}) = CheckCustomer(o),\ \mathcal{N}(1, 0.8; t);\ A(\vec{x}) = VerifyPayment(o),\ \mathcal{N}(1, 1; t);\ A(\vec{x}) = ChargeMoney(o),\ \mathcal{N}(2, 2; t)]$

where $\mathcal{N}(\mu, \sigma; t)$ is a probability density over time $t > 0$ of form Gaussian with mean μ and standard deviation σ.

Define the notation for the total expected cost as:

$$tcCase(A(\vec{x})) = kCase(A(\vec{x})) \oplus (cCase(A(\vec{x})) \otimes \int_0^{T_{max}} e^{-\alpha t} fCase(A(\vec{x}))dt) = case[\beta_1(A(\vec{x})), (k_1 + c_1 \int_0^{T_{max}} e^{-\alpha t} f_1(t)dt); \ldots; \beta_{|A|}(A(\vec{x})), (k_{|A|} + c_{|A|} \int_0^{T_{max}} e^{-\alpha t} f_{|A|}(t)dt]$$

The case notation for the total expected reward, Eq. 3.2, becomes:

$$rCase(s, A(\vec{x})) = rsCase(s) \ \ominus \ tcCase(A(\vec{x})) \tag{4.1}$$

Next, we define the case notation for Eq. 3.4:

$$mCase(n_j(\vec{x}), A(\vec{x}), s) = \int_0^{T_{max}} e^{-\alpha t} pCase(n_j(\vec{x}), A(\vec{x}), s) \otimes fCase(A(\vec{x}))dt \tag{4.2}$$

where $n_j(\vec{x})$ is a deterministic decomposition of the stochastic action, $A(\vec{x})$. Thus there are as many such cases as the number of deterministic actions.

Given Eqs. 4.1 and 4.2, we may solve FO-SMDPs using *symbolic value iteration*, in a manner similar to FO-MDPs (Section 3.4). Specifically, we replace the $rCase$ and $pCase$ in Eq. 3.5 with $trCase$ and $mCase$, respectively.

4.2 Model Elicitation from Web Service Descriptions

We briefly mention ways in which the model parameters of the above mentioned primitive FO-SMDP are obtained. The actions, $A(\vec{x})$, are the atomic operations in WSs that compose the WSC. Preconditions for performing the actions are directly obtained from the preconditions of WSs specified using RuleML, in their OWL-S or SAWSDL descriptions. Successor state axioms are compiled from the first order effect sentences in the WS descriptions. For example, consider the description of the following WS operation:

Web service operation: $ChargeMoney(o)$
Precondition: $ValidCustomer(o)$ AND $ValidPayment(o)$
Effect: $Charged(o)$

The precondition axiom for action $ChargeMoney(o)$ is:

$$ValidCustomer(o, s) \wedge ValidPayment(o, s) \Rightarrow Poss(ChargeMoney(o), s)$$

The successor state axiom becomes:

$$Poss(a, s) \Rightarrow Charged(o, do(a, s)) \Leftrightarrow a = ChargeMoneyS(o) \vee Charged(o, s)$$

Some examples of compiling successor state axioms from DAML-S [29] WS descriptions can be found in [53]. The probabilities of the different responses or effects from service invocations that make up the probabilities in *pCase* may be found in either the *serviceParameter* section of the OWL-S description of the WS or in the *SLAparameter* section of the WSLA specification [48](see Fig. 4.2). These probabilities quantify contracted service reliability rates.

```
<ServiceLevelObjective name="InventoryAvailabilityRate">
  <Expression>
    <Predicate xsi:type="Equal">
      <SLAParameter>InventoryAvailability</SLAParameter>
      <Value>0.4</Value>
    </Predicate>
  </Expression>
  ......
<ServiceLevelObjective>
```

Figure 4.2: A WSLA snippet illustrating the specification of inventory availability rate.

The costs in $kCase$, which represents the parameter, K, may also be obtained from the *serviceParameter* section of the OWL-S description or from the agreement between the service users and providers. The values in the case notation of the sojourn time distribution, F, and the cost rate, C, are typically selected by the system designer from past experience.

4.3 Composite FO-SMDPs

For the lowest levels of the WSC, Haley uses the FO-SMDP, defined in Section **??** to model the composition problem. Let us label these FO-SMDPs as *primitive*. In primitive FO-SMDPs, actions are WS invocations, and sojourn times are the response times of the WSs. We compose the higher levels of the WSC using a composite FO-SMDP (C-FOSMDP). Within a

C-FOSMDP, the actions are either *abstract* and represent lower level WSCs which in turn are modeled using either composite or primitive FO-SMDPs, or simple WS invocations. For example, *VerifyOrder* in the supply chain example (Section 6.1) is modeled as an abstract action because it represents a lower level process composed of three actions *CheckCustomer*, *VerifyPayment* and *ChargeMoney*, each of which is a primitive WS invocation.

We use a to represent a primitive action and $\bar{a}$ to represent an abstract action. The elicitation of the C-FOSMDP model parameters contingent on primitive actions is similar to that of the primitive FO-SMDP as shown in Section 4.2. However, model parameters for abstract actions are not directly available and must be *derived* from the model parameters of the corresponding primitive FO-SMDP that models the lower level WSC.

For the sake of simplicity, we focus on deriving the model parameters for a composition that is singly-nested. Our methods generalize to a multiply-nested composition in a straightforward manner. We utilize the correspondence between the high-level abstract action and the corresponding low-level primitive actions. For illustration, we take the abstract action $VerifyOrder(o)$ as the example to explain how we derive the logical representations of the model parameters for abstract actions. Specifically, in addition to the successor state axioms, we need to derive the $pCase$, $kCase$, $cCase$, and $fCase$, for the abstract action.

While the underlying methods for computing the parameters for the abstract action are the same as in [104], we adapt them to the use of case notation. Thus, we will add a new case in each of the case notations of the C-FOSMDP for the abstract action.

• $pCase$ **statements for abstract action:** As $VerifyOrder(o)$ is a stochastic action, we decompose it into two deterministic actions, each representing the nature's choice. Let $VerifyOrderS(o)$ and $VerifyOrderF(o)$ be nature's choices denoting a validated and failed order, respectively. Notice that for the order to be valid, the customer and payment should be valid and the money charged. Thus,

$$Pr(VerifyOrderS(o), VerifyOrder(o), \bar{s}) = Pr(CheckCustomerS(o), CheckCustomer(o), s) \times$$

$Pr(VerifyPaymentS(o), VerifyPayment(o), s) \times Pr(ChargeMoneyS(o), ChargeMoney(o), s)$
$= 0.9 \times 0.8 \times 0.98 = 0.71$

$Pr(VerifyOrderF(o), VerifyOrder(o), \bar{s}) = 1$ - $Pr(VerifyOrderS(o), VerifyOrder(o), \bar{s}) = 0.29$

These probabilities are added as cases in the $pCase$ for the C-FOSMDP:

$pCase(VerifyOrderS(o), VerifyOrder(o), \bar{s}) = [true; 0.71]$
$pCase(VerifyOrderF(o), VerifyOrder(o), \bar{s}) = [true; 0.29]$

• **Successor state axiom for abstract action:** Recall that a successor state axiom describes the effect of an action on a fluent. Let $ValidOrder(o, \bar{s})$ be the fluent affected by $VerifyOrder(o)$. Then,

$Poss(\bar{a}, \bar{s}) \Rightarrow ValidOrder(o, do(\bar{a}, \bar{s})) \Leftrightarrow \bar{a} = VerifyOrderS(o) \vee ValidOrder(o, \bar{s})$

In order to ground the successor state axiom for $VerifyOrder(o)$, we note the following relationship between the fluent, $ValidOrder(o, \bar{s})$ and the fluents of the corresponding primitive actions:

$ValidOrder(o, \bar{s}) \equiv ValidCustomer(o, s_1) \wedge ValidPayment(o, s_2) \wedge Charged(o, s_3)$

In addition, as we mentioned before,

$VerifyOrderS(o) \equiv CheckCustomerS(o) \wedge VerifyPaymentS(o) \wedge ChargeMoneyS(o)$

The successor state axiom for $VerifyOrder(o)$ becomes:

$Poss(\bar{a}, \bar{s}) \Rightarrow ValidOrder(o, do(\bar{a}, \bar{s})) \Leftrightarrow [a = CheckCustomerS(o) \wedge a = VerifyPaymentS(o) \wedge a = ChargeMoneyS(o)] \vee [ValidCustomer(o, s_1) \wedge ValidPayment(o, s_2) \wedge Charged(o, s_3)]$

• $kCase$ **statement for abstract action:** The lump sum cost of an abstract action is a summation of the lump sum costs of the associated low-level primitive actions:

$k_{VO} = kCase(CheckCustomer(o)) + kCase(VerifyPayment(o)) + kCase(ChargeMoney(o))$

We add a statement to the $kCase$ of the C-FOSMDP: $(\bar{A}(\vec{x}) = VerifyOrder(o),\ k_{VO})$

• $fCase$ **statement for abstract action:** Let the sojourn times of the low-level primitive actions follow Gaussian distributions with means μ_{CC}, μ_{VP}, and μ_{CM}, and corresponding standard deviations σ_{CC}, σ_{VP}, and σ_{CM}. The sojourn time distribution of the abstract action $VerifyOrder(o)$ also follows a Gaussian defined as: $f_{VO}(t) = \mathcal{N}(\mu_{VO}, \sigma_{VO}; t)$ where: $\mu_{VO} = \mu_{CC} + \mu_{VP} + \mu_{CM}$ and $\sigma_{VO} = \sqrt{\sigma_{CC}^2 + \sigma_{VP}^2 + \sigma_{CM}^2}$

We add a statement to the $fCase$ of the C-FOSMDP: $(\bar{A}(\vec{x}) = VerifyOrder(o),\ f_{VO}(t))$

• $cCase$ **statement for abstract action:** We note that the accumulated cost of an abstract action is the total accumulated cost of all the corresponding primitive actions. Using the sojourn time distributions of the primitive actions, we compute the expected sojourn time E_{a_i} of each, and use it to derive the rate:

$c_{VO} = \frac{cCase(CheckCustomer(o)) \times E_{CC} + cCase(VerifyPayment(o)) \times E_{VP} + cCase(ChargeMoney(o)) \times E_{CM}}{E_{CC} + E_{VP} + E_{CM}}$

where: $E_{a_i} = \int_0^{T_{max}} t f(t|a_i) dt$; a_i is the primitive action; $f(t|a_i)$ is the sojourn distribution

of the primitive action.We add the following statement to the $cCase$ of the C-FOSMDP: $(\bar{A}(\vec{x}) = VerifyOrder(o),\ c_{VO})$

After deriving the logical representations for abstract actions, the C-FOSMDP is well defined and may be solved just like a primitive FO-SMDP using the symbolic value iteration as mentioned previously. By providing general methods for deriving the C-FOSMDP model parameters from those of the lower level ones, we allow C-FOSMDPs at any level to be formulated and solved using the standard solution methods.

Having described the theoretical framework, we present the architecture and modules of our implementation of Haley as a tool suite to support WSC.

4.4 Composition Generation and Execution

Solving the C-FOSMDPs and primitive FO-SMDPs defined previously generates a policy at each level of the hierarchy. A policy, π, itself in case notation, $\pi Case$, maps first order sentences, which represent regions of the state space where the sentences are true, to WS invocation(s). The action is expected to be optimal over the period of consideration. Our policy based approach of generating a WS composition is robust – no matter what the outcome of the WS invocation is, the policy will prescribe the next WS to invoke.

4.4.1 Algorithm

Haley generates and executes a WSC top-down, using the policy prescriptively to guide the selection of the next WS to invoke. If the policy prescribes an abstract action, Haley utilizes the policy and start state of the lower level WSC. In order to generate the composition, we need a way to find out which of the case conditions in the policy is entailed at each step. We do this by maintaining a first order logic based KB implemented in Prolog. The KB is initialized using the initial state of the high level WSC. Using the ASK operator, the KB is queried to find out which of the case conditions in the corresponding $\pi Case$ is

entailed. [1] Given the entailed case statement, the WS prescribed by the policy is invoked and its responses interpreted as effects update the KB using the TELL operator. We additionally tell the KB that the action representing the WS has been performed. This procedure is repeated until the KB entails the terminal condition or the specified number of steps have been performed.

We deploy the higher level policy as a WS-BPEL composition and wrap the KB as a WS. Each of the lower level policies is described using WS-BPEL files of their own. The preconditions and effects of the WSs are described using first order RuleML. Haley's algorithm for generating and executing WS compositions is shown in Fig. 4.3.

[1]Note that only a single case condition will be entailed because the case statements form a partition of the state space.

Algorithm for Composing and Executing Nested Web Process

Input: $\pi Case$ *//policy in case notation form*
s_0 *//logical description of the initial state of Web process*

$s \leftarrow s_0$
$initialize(KB, s_0)$ *//initialize KB with the initial state*
while KB $\not\models$ logical description of the terminal states
for each case statement Ψ_i in policy case notation $\pi Case$
if ASK(KB, Ψ_i)
$s \leftarrow \Psi_i$ *//s is the case statement entailed by KB*
break
end if
end for
$a \leftarrow \pi Case(s)$ *// a is the optimal action*
if a is a primitive action **then**
Invoke WS representing a and get response of WS
TELL(KB, Effect(a)) *//Update KB with effect of invocation*
else *//a is an abstract action*
$s_{initial} \leftarrow$ initial state of the lower level process
$\pi' Case \leftarrow$ corresponding policy case notation
Recursively call this algorithm with $\pi' Case$, $s_{initial}$
Get response of the lower level Web process
TELL(KB, Effect(a)) *//Update KB with effect of invocation*
end if
end while
if $\pi Case$ is not the policy for the top-level FO-SMDP **then**
return invocation response
end if
end algorithm

Figure 4.3: Interleaved composition and execution of a nested WSC in Haley.

Chapter 5

Implementation and Performance Study

Haley is implemented as a suite of freely-available Eclipse [1] plug-ins and a stand-alone Eclipse rich client platform (RCP) application. It is provided under the Eclipse public license version 1.0. [2] Some of the technologies used in developing Haley are Draw2d, Eclipse modeling framework (EMF) [3], graphical modeling framework (GMF) [2], Prolog and ActiveBPEL API [3]. Haley also contributes several independent tools and experiences to the SOA community: (*i*) SAWSDL (semantic annotations for WSDL) viewer is a complete and independent Eclipse plug-in and is the first viewer for SAWSDL files. (*ii*) eDT-GOLOG may be used as a general stand-alone decision-theoretic planner. The system is compliant with the Eclipse plug-in standards and can be integrated with other Eclipse based tools like Web tools platform (WTP), WSDL editor, and ActiveBPEL simulator and designer in case process designers want to customize the generated BPEL code.

5.1 Architecture

Haley is composed of four major components:

1. *WS and goal specification* This component is responsible for parsing service description files including SAWSDL and WSLA (WS level agreements) files. It also provides ways for the process designer to specify the WSC hierarchy and goal.

[1] Eclipse platform: http://www.eclipse.org
[2] Haley is available for download at **http://denali.cs.uga.edu/haley**
[3] ActiveBPEL: http://www.activevos.com/bpel.php

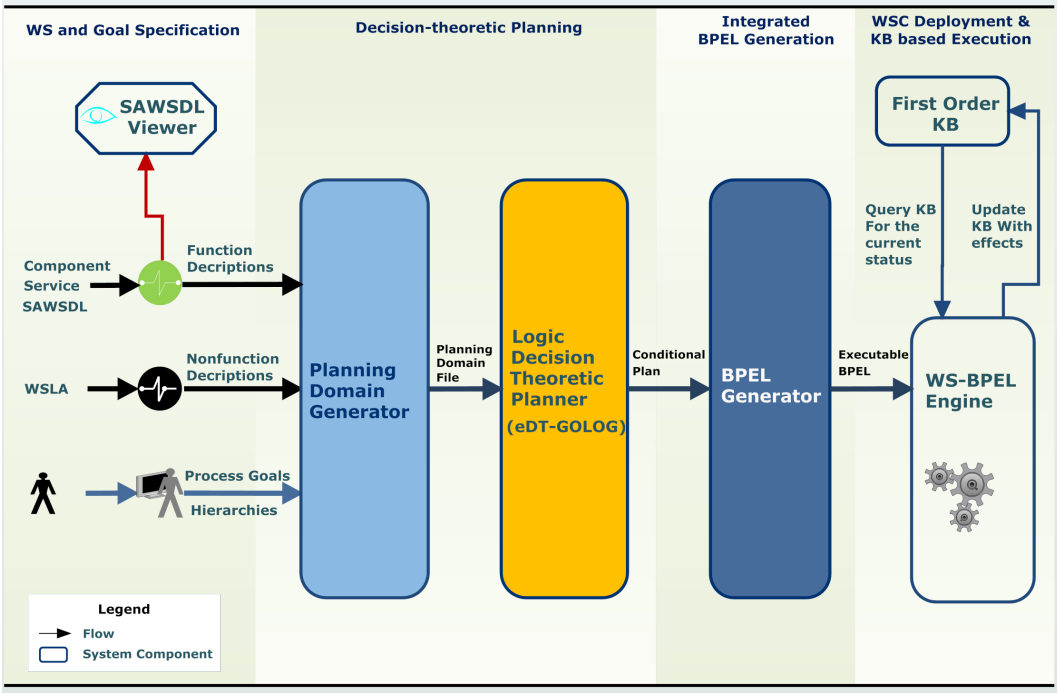

Figure 5.1: Architectural details of Haley. Notice that Haley processes both service descriptions and agreement specifications. Information from these files is used to formulate the planning problem (often called the planning domain) automatically.

2. *Decision-theoretic planning* This component is responsible for producing a planning problem formulation from the information gathered by the previous component. It generates a policy using the decision-theoretic planner.

3. *Integrated BPEL generation* This component transforms the generated policy into executable BPEL code; and

4. *WSC deployment and KB based execution* This component is responsible for deploying the generated BPEL and monitoring the execution. We show the architecture in Fig. 5.1 and further describe the main components of Haley below.

5.2 Modules

The modularized architecture of Haley enables support for future improvements and extensions. For example, Haley could support other types of service description specifications such as OWL-S and WS-Agreement by simply plugging new parsing modules for these descriptions. We describe the current modules individually:

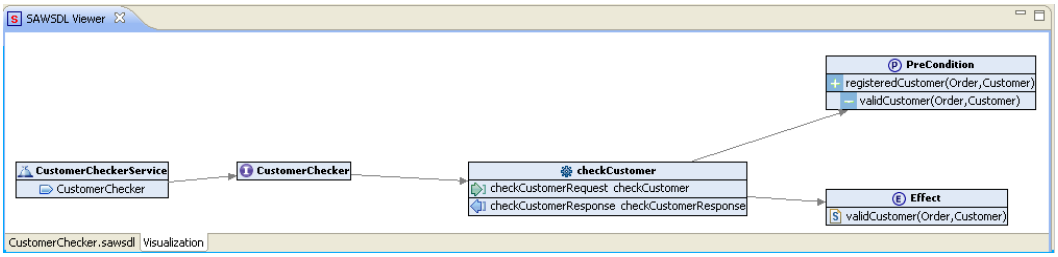

Figure 5.2: SAWSDL viewer showing an example SAWSDL described WS *CheckCustomer*.

• **SAWSDL Parser and Viewer** SAWSDL extends WSDL by allowing semantic annotations in the form of model references and schema mappings. In addition to specifying the inputs and outputs of a service, SAWSDL also allows the specification of preconditions and effects, which are useful for composing services. However, the current SAWSDL specification does not ground preconditions and effects using any language. We therefore extend SAWSDL to support preconditions and effects specified using SWRL, a popular semantic Web rule language. [4] In addition to parsing SAWSDL using the SAWSDL4J API, Haley provides a new Eclipse plug-in for graphically viewing SAWSDL based service descriptions. We show a snapshot of the viewer in Fig. 5.2.

• **WSLA Parser** Web service level agreements (WSLA) specify the non-functional quality of service (QoS) parameters of WSs such as response times, costs and availability percentages. Haley is not limited to any particular service agreement specification and can be easily extended to support WS-Agreement or other agreement specifications. QoS considerations are often neglected while manually designing BPEL processes as well as by many other automated composition techniques. Haley uses a decision-theoretic planner for the composition

[4] SWRL: http://www.w3.org/Submission/SWRL

that provides an intuitive way to model the QoS parameters and optimize them over the long term.

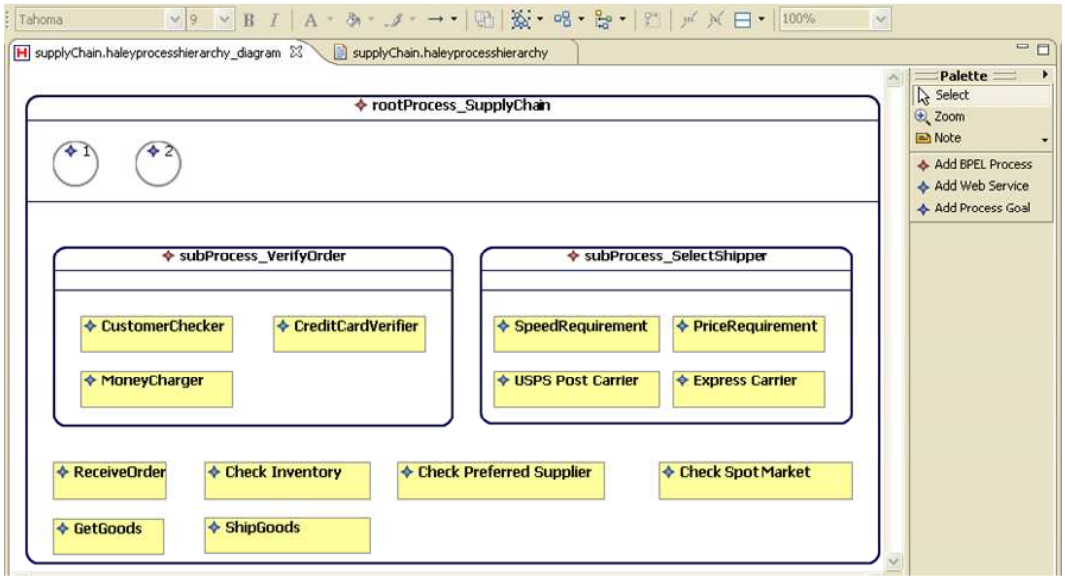

Figure 5.3: Hierarchy Modeler with a GUI for intuitively grouping together the WSs participating in the composition into a hierarchy.

• **Process Hierarchy Modeler** Service composition methods often do not scale well to many services, which makes it difficult to use them in real-world applications. Haley promotes scalability by exploiting the hierarchies usually found in real-world processes. To facilitate this, Haley provides an intuitive GUI (see Fig. 5.3) to construct a WSC hierarchy by importing component WSs at each level. The modeler may also be used to specify the start states, multiple goals and associated priorities at each level of the hierarchy. All of this information is written into an XML file for input to the planner.

• **Planning Domain Generator** Given the functional and non-functional descriptions of individual WSs and goal descriptions for the target composition, we automatically generate a corresponding planning problem domain file. The planning problem contains a first order logical description of the operations and their inputs, outputs, preconditions and effects as well as the goals.

• **eDT-GOLOG Planner** As an extension of DT-GOLOG [19, 87], we designed eDT-GOLOG to support first-order SMDP based decision-theoretic planning which is applicable

to situation calculus and decision theory. In addition to being expressive, eDT-GOLOG allows us to model the uncertainty of WS operations, QoS measures and provide guarantees of optimality while preserving efficiency of planning as much as possible. The planner takes as input the planning domain file and produces a policy or a conditional plan for the composition.

• **BPEL Generator** Haley transforms the conditional plan output by the planner into an executable WS-BPEL file. Manually designing a BPEL process requires designers to specify namespaces, variables, partner links, and the control flow of the activities. Haley programmatically generates an executable BPEL using the ActiveBPEL API and deploys it in an ActiveBPEL engine. This saves time and effort, and avoids common grammatical and logical errors while designing BPEL processes.

• **KB based Process Monitor** In order to determine which branches to take while executing the BPEL, service operations once performed are asserted in a first order logic KB. The KB is implemented using an embedded Prolog engine wrapped in a WS. The KB is updated with the effects of the operations and queried for the next operation to perform.

In summary, Haley automatically composes an executable WS-BPEL process given component services described using SAWSDL and service agreement files, using a first-order logic based planner that is both scalable and expressive.

5.3 Performance Evaluation

We empirically evaluated the performance of Haley in comparison with two other well-known WS composition techniques: HTNs augmented with information gathering actions [99] and MBP [73] (used in the Astro project). We performed the evaluation on the two application scenarios mentioned in Section 6.1. In Figs. 5.4 and 5.5, we show the average rewards obtained by executing the WSCs using each of the three approaches as we vary the uncertainty of the composition environment. For our experiments, we varied the non-functional parameter, availability, of the USPS WS in Fig. 5.4, and the probability with which the

inventory satisfies the manufacturer's order in Fig. 5.5 . Each data point is the average of 1000 executions of the composition where each execution involves running the WSC until the compositions are successfully completed or it is unable to move forward.

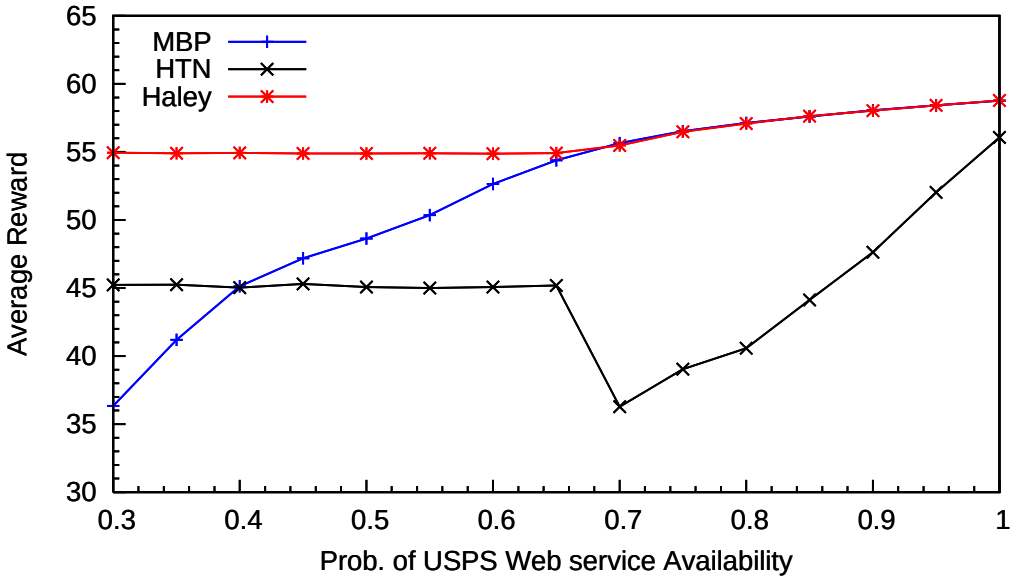

Figure 5.4: Average rewards on running the compositions generated by HTN, MBP and Haley for the online shopping example. Haley gathers the most reward because it models the non-determinism of WSs and provides a cost-based composition optimization. Performances of all the approaches begin to converge as the availability approaches 1 signifying that the uncertainty reduces.

For the online shopping application scenario (Fig. 5.4), we observe that the composition generated by HTN performs the worse. HTN-generated WSC performs poorly because the execution of the composition stops prematurely when the WS is unavailable to take requests. For lower rates of WS availability, this happens frequently, and is responsible for the lower average reward of the composition. We observe that there is a sudden decrease of the average reward when the probability of USPS availability is around 0.68. This is where the change in the optimal choice occurs. As the availability of USPS WS increases, the optimal choice becomes the USPS WS as opposed to the FEDEX WS. Although the cost of using USPS is lower than using FEDEX, the FEDEX WS availability is still higher than the USPS WS availability at this point. Hence, the overall reward of the process is lower. The MBP-generated process performs better because its execution, similar to Haley, is also guided by

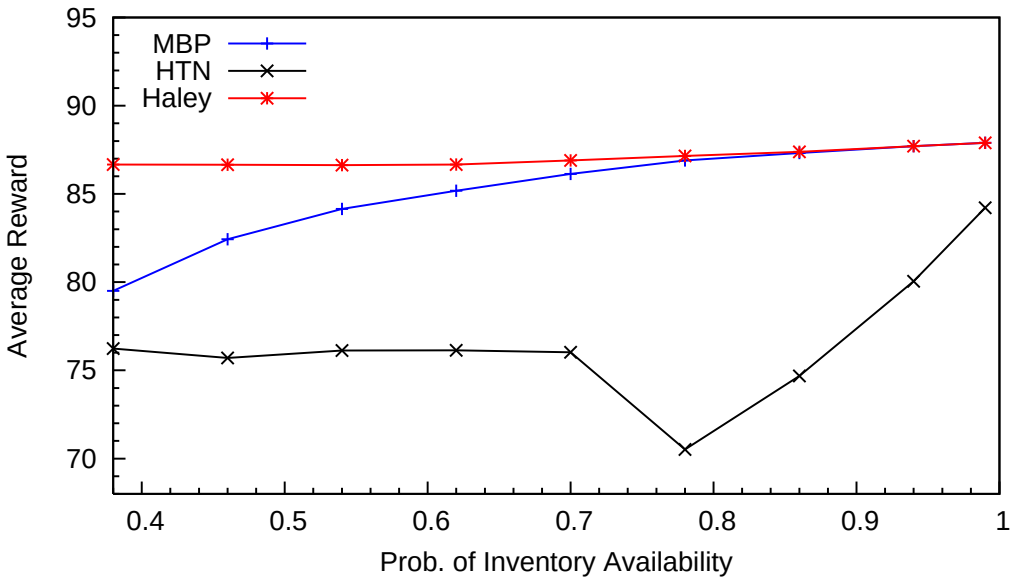

Figure 5.5: Average rewards on running the processes generated by the HTN, MBP and Haley for the supply chain example. It demonstrates similar behaviors of three approaches as seen in Fig. 5.4

a policy [28]. However, even at low probabilities of USPS WS availability the composition invokes the USPS for satisfying the order. This is because MBP does not associate costs with actions and fails to distinguish between candidate WSs of similar functionality (USPS and FedEx) but with different non-functional parameters. Haley chooses to bypass USPS and utilizes FedEx, which is responsible for its better performance. As the USPS WS availability improves, Haley switches to checking inventory and its performance becomes close to that of the MBP.

Analogous to the online shopping problem, in the order handling application scenario, we observe similar behaviors (Fig. 5.5). HTN-generated WSC performs the worse because the execution of the composition often stops prematurely when the inventory or the preferred supplier is unable to satisfy the order. This could happen when the WS is not functioning or there are not enough products to satisfy the order. As the inventory availability improves, it becomes the optimal choice. The lower reward is due to its availability being lower than

the preferred supplier at this point. The MBP-generated process performs better because its execution is guided by a policy. However, it is not able to distinguish between candidate WSs with similar functionality but different QoS parameters. Thus, even at low probabilities of inventory availability the composition repeatedly invokes the inventory for satisfying the order. Haley chooses to bypass the inventory and utilizes the preferred supplier, which is responsible for its better performance. As the inventory availability improves, Haley switches to checking inventory and its performance becomes close to that of the MBP and HTNs.

The performance of the WSC approaches is determined by the capability of the planner. HTN planning does not model uncertainties in WSs, nor does it associate costs with planning states or actions. MBP is capable of planning with non-deterministic actions, but it does not associate probabilities with different outcomes and it does not associate costs with planning states or actions either. Therefore, both these approaches have limited ability to perform optimization during the planning phase.

In Table 5.1, we demonstrate the advantages of a hierarchical decomposition and logic based representation using the time taken in generating the plans. We compare between approaches that utilize a hierarchy such as the hierarchical formulation for SMDPs [104] and Haley and their counterparts that do not. We also compare between approaches that utilize a logical representation such as FO-SMDP and Haley and those that utilize the traditional propositional state space representation.

For the first application scenario of online shopping, hierarchical approaches consume significantly less time than their flat counterparts. For the second scenario, we also varied the number of distinct types of orders handled by the supply chain to see how first-order logic representation reduces the size of the state space. Different types of orders differ in the probabilities of fulfilling them and their costs. In propositional approaches, a distinct order type would be included as a new proposition in the state space. In first order logic, this is equivalent to grounding the variable o in the predicates with the corresponding number of the type. We observe that the flat SMDP whose states are obtained by grounding and

Table 5.1: Run times for generating policies that guide the compositions (Centrino 1.6GHz, 512MB, WinXP). Flat FO-SMDP and Haley perform better than propositional SMDP and propositional hierarchical SMDP as a result of using first-order representations. Hierarchical SMDP and Haley have better run times than their corresponding flat frameworks. This is because the hierarchical decomposition significantly reduces the planning state space.

Scenarios		**Flat SMDP**	**Hier. SMDP**	**Flat FO-SMDP**	**Haley**
Online shopping		251.49s	0.241s	34.19s	0.54s
Order handling	Order types				
	1	741.83s	0.46s	82.95s	0.93s
	2	*	1.5s	82.95s	0.93s
	3	*	15.27s	82.95s	0.93s
	5	*	695.02s	82.95s	0.93s
Order handling with 15 suppliers		*	*	3183.29s	159.67s

propositionalizing is computationally most expensive. This is because its state space grows exponentially as the supported number of order types increases. In contrast, FO-SMDP takes significantly less time. In both scenarios, the effectiveness of the hierarchical decomposition is evident from the fact that the hierarchical approaches consumed significantly less time than the others. This is because the decomposition leads to smaller state spaces, and the planning at the different levels could occur in parallel.

We further tested scalability by increasing the number of suppliers in the order handling scenario to 15, resulting in a total of 24 WSs in the composition. As we may expect, propositional approaches failed to generate a solution in a reasonable amount of time. While FO-SMDP generated a plan, it took an order of magnitude time greater than Haley in doing so.

Finally, we show the execution times of running the WS-BPEL based compositions generated by Haley and the other approaches, in Table 5.2. All WSs were deployed in an Axis 1.2 implementation, while the WS-BPEL files were executed using the ActiveBPEL engine. The executions times of the hierarchical approaches are greater than the flat approaches due

Table 5.2: Execution times of the WS-BPEL compositions averaged over 100 runs (Centrino 1.6GHz, 512MB, WinXP).

Scenarios	Flat SMDP	Hier. SMDP	Flat FO-SMDP	Haley
Online shopping	240.95ms ± 62.1ms	420.25ms ± 76.4ms	429.86ms ± 69.4ms	1137.94ms ± 138.4ms
Order handling	255.52ms ± 109.2ms	335.05ms ± 60ms	436.49ms ± 109ms	970.88ms ± 136.1ms

to the overhead incurred while invoking the lower level WS-BPEL compositions. In comparison, the flat WS-BPEL files invoke WSs only. The algorithm for interleaved composition and execution in Haley (Fig. 4.3) requires the use of ASK statements on a first order logic KB. While in the worst case this could be semi-decidable [82, 89], the execution times for Haley demonstrate that the ASK statements typically entail simple and quick inferences in practice. However, as we may expect, interactions with the KB lead to execution times that are greater than those of the traditional propositional approaches.

In summary, Haley generates WS compositions significantly faster than comparative, traditional propositional approaches. Furthermore, it is able to compose processes much larger while simultaneously modeling the uncertainty of WSs and optimizing QoS parameters. On the other hand, executing the compositions takes longer due to the interleaved interaction with the KB for determining the logical state of the composition and applicable actions.

Chapter 6

RESTful Web Service Composition

We have covered our proposed approach to automated WSDL/SOAP WS composition in the previous chapters. In this Chapter, we discuss another paradigm of building WSs, RESTful WSs, and present our approach to the automated RESTful WS composition problem accordingly.

Emerging as the popular choice for leading Internet companies to expose internal data and resources, RESTful WSs are attracting increasing attention in the industry. While automating WSDL/SOAP based Web service composition has been extensively studied in the research community, automated RESTful Web service composition in the context of SOA is less explored. Due to the differences between WSDL/SOAP bases WSs and RESTful WSs, the composition problem of these two types of WSs are fundamentally different.

As early effort addressing this problem, we discusses, in this chapter, the challenges of composing RESTful WSs and propose a formal model for describing individual WSs and automating the composition. It demonstrates our approach by applying it to a real-world RESTful Web service composition problem.

6.1 Motivating Scenario

In this section, we introduce a simplified online shopping scenario. Registered customers place orders to the system, and one customer may have multiple orders in the system. Orders are not handled until the payment is received. Once the payment is verified, the system processes the order and ships the order to the customer. This system is intended to be implemented

using WSs so that it can be used by external business partners or third party Web 2.0 applications. We would like to expose each of the functional components using WSs.

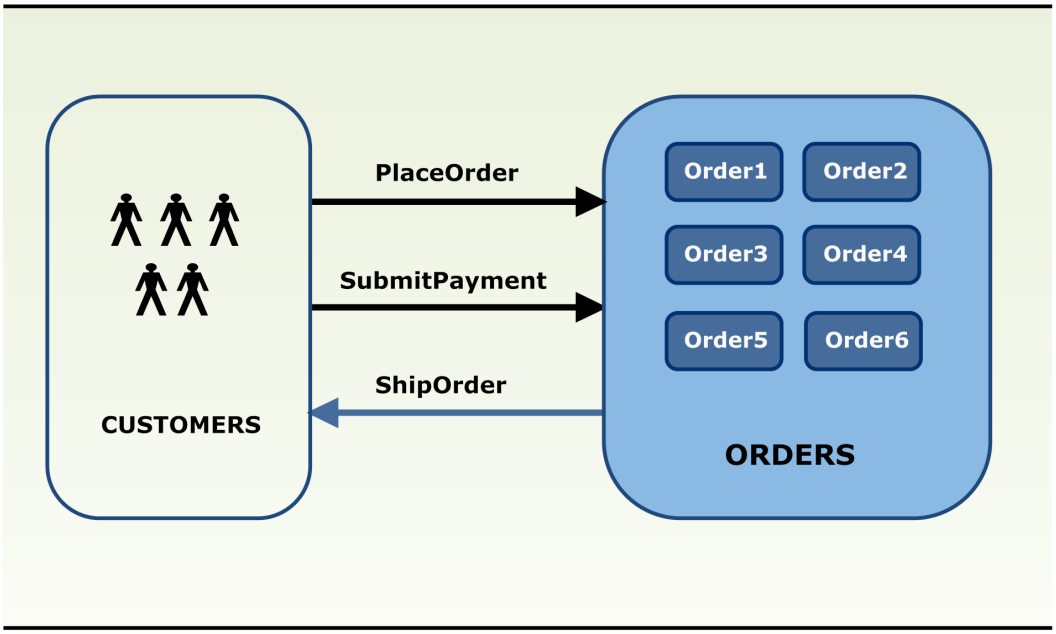

Figure 6.1: A simplified online shopping scenario

An imperative approach of handling this application would start with functionality decomposition. One solution could be to expose Remote Procedure Call (RPC) style WSs as below:

6.2 Modeling RESTful Web Service

In this section, we introduce a classification of RESTful WSs, and present a conceptual modeling approach to describe the identified types of WSs. Unlike WSDL/SOAP based WSs, there is no commonly recognized model or description language available for RESTful WSs. To facilitate automated composition, we present an ontology based formal model for RESTful WSs, this model is at the conceptual level and may be bounded with ontology languages.

Table 6.1: The list of RPC style WSs

WSDL/SOAP WSs and Interfaces		
getCustomer	Description:	get customer informat
	Input:	customer id
	Output:	customer information
getOrder	Description:	get order information
	input:	order id
	output:	order information
placeOrder	Description:	place a new order
	Input:	customer id, order
	Output:	success or failure
SubmitPayment	Description:	submit a payment
	Input:	order id, payment
	Output:	success or failure
ShipOrder	Description:	ship the order
	Input:	order id
	Output:	success or failure

6.2.1 Classification of RESTful WSs

Most resources associated with RESTful services can be directly mapped to domain resources – either a set of resources or individual resources. Besides these two types, we identify a third type of RESTful WSs – these services consume some resources or manipulate related resources, they can not be directly mapped to domain resources or resource collections. We call them *transitional* RESTful WSs. This type of RESTful WSs are less declarative than the other two types, and we should minimize the use of this type of services when we adopt a resource-centric approach to design WSs. But in some cases, we do need transitional RESTful WSs as we show using application scenario mentioned in Section 6.1.

Type I: Resource Set Service This type of services is mapped to a set of domain resources. In the online shopping scenario, resource related to a set of customers and a set of orders can be both considered of this type. We name them CustomerSet and OrderSet respectively. Type I RESTful Web service may be utilized to capture the concept level

resources or the set of instance resources. This type of services support all four HTTP operations(GET, PUT, DELETE and POST).

Type II: Individual Resource Service Individual domain resources can be modeled with this type of services that represent the individual resources in the resource set. For example, in the scenario, individual customer and individual order are mapped to this type. Individual payment and shipment are also considered of this type associated with orders and customers. Type II RESTful Web service may be utilized to capture instance level resources, and it supports three HTTP operations(GET, PUT, DELETE). Operation POST is not applicable here since the URI identified individual resource is already created.

Type III: Transitional Service Although most of RESTful WSs are mapped to the domain resources or resource sets, some of the services are more transition or transformation oriented. The functionality of this type of services is loosely defined as services that consume resources, create resources and update or transform the states of the related resources. For example, SubmitPayment and ShipOrder in the scenario are of this type. When we invoke SubmitPayment with the order information, we create a new resource payment associated with this order, and we update the isPaid property, denoting the payment status, of the order resource from "false" to "true". Similar steps need to be done for the Web service ShipOrder as well. Type III Web service may be utilized to capture transition-oriented functionalities, and it only supports POST operation.

In the next sub-section, we provide a detailed modeling approach to describe these types of WSs identified above.

6.2.2 Modeling RESTful WSs

By identifying these types of RESTful WSs, we take a resource-centric look at the original scenario. A list of the declarative WSs needed to model the online shopping scenario are presented in Figure 6.2. In the rest of this section, we propose an ontology based conceptual

model to describe these identified RESTful WSs. This model will be used to build our automated composition framework in Section 6.3.

Figure 6.2: Identified RESTful WSs

RESTful WSs		
CustomerSet	TYPE:	Type I
	URI:	http://some.com/Customers
	Supported Operations:	GET, PUT, DELETE, POST
OrderSet	TYPE:	Type I
	URI:	http://some.com/Customers/[Customer_id]/orders
	Supported Operations:	GET, PUT, DELETE, POST
Customer	TYPE:	Type II
	URI:	http://some.com/Customers/[Customer_id]
	Supported Operations:	GET, PUT, DELETE
Order	TYPE:	Type II
	URI:	http://some.com/customers/[customer_id]/orders/[order_id]
	Supported Operations:	GET, PUT, DELETE
Payment	TYPE:	Type II
	URI:	http://some.com/customers/[customer_id]/orders/[order_id]/payment
	Supported Operations:	GET, PUT, DELETE
Shipment	TYPE:	Type III
	URI:	http://some.com/customers/[customer_id]/orders/[order_id]/shipment
	Supported Operations:	POST
SubmitPayment	TYPE:	Type III
	URI:	http://some.com/customers/[customer_id]/orders/[order_id]/submitpayment
	Supported Operations:	POST
ShipOrder	TYPE:	Type III
	URI:	http://some.com/customers/[customer_id]/orders/[order_id]/shipOrder
	Supported Operations:	POST

To expose resources as RESTful WSs for a particular domain, it is intuitive to create the association between Web service resources and domain ontology resources. The idea is that a declarative RESTful approach models each service as a resource, so the description of WSs is essentially the description of the resources and the "state transfer" of these resources. As we classify RESTful WSs into three classes, we explain the specific modeling for them respectively. Generally speaking,

(1) Type I RESTful Web service is a set of ontology instances of the same concept, and the "set" itself could be also considered as a resource as well. While applying GET, DELETE and PUT operations to Type I RESTful Web service, it will fetch, remove and update the representation of this concept resource respectively; Applying POST operation will add into the resource set a new instance resource of this concept.

(2) Type II RESTful Web service is directly mapped to an ontology instance. Type I and Type II RESTful WSs are modeled directly using mapped resources in ontology. Applying GET, DELETE and PUT operations to Type II RESTful Web service will fetch, remove and update the representation of the corresponding instance resource based on the standard semantics of these HTTP operations. POST operation is not applicable for Type II service.

(3) Type III RESTful Web service is transition-oriented, and we describe them as "state transfer" of resources using transition rules. We adopt Semantic Web Rule Language (SWRL) [41] to formally describe these rules. SWRL is a formal language to describe rules based on OWL and RuleML. It has been widely used to describe semantic rules in the ontology context. In our modeling framework, services of Type I and II are mapped to ontology resources; services of Type III are described by the transition rules of ontology resources. Consequently, SWRL becomes an appropriate choice to describe the rules associated with Type III RESTful WSs.

To define the formal semantics of these three types of WSs, we define two new classes: WSRESOURCE, RESTWS. WSRESOURCE describes the resources RESTful WSs represent. WSRESOURCE is either mapped to an individual ontology instance (Type II) or a set

Table 6.2: WSRESOURCE Definition

Def WSRESOURCE:	
has_name:	onto:wsresource#name
has_description:	onto:wsresource#description
map:	onto:set(onto:resource)

Table 6.3: Type I RESTful Web Service

Def RESTWS I:	
has_name:	onto:restws#ws-name
has_description:	onto:restws#description
has_URI:	onto:restws#uri
has_type:	onto:restws#type
has_wsresource:	onto:wsresource
onGET:	onto:opn:Supported
onPUT:	onto:opn:Supported
onDELETE:	onto:opn:Supported
onPOST:	onto:opn:Supported

of ontology instances (Type I). RESTWS is used to describe the RESTful WSs, RESTWS contains its associated WSRESOURCE. Type I & II RESTful WSs has only one associated WSRESOURCE, Type III RESTful Web service may have multiple associated WSRESOURCEs.

We describe the formal semantics of the universal HTTP operations (GET, POST, PUT and DELETE) on these three types of RESTful WSs. We use *onto* as the name space for the associated domain ontology, and *swrl* as the name space for SWRL.

Modeling Type I Web service Type I Web service is mapped to a set of instances of the same concept in ontology.

Table 6.4: WSRESOURCE-ORDERSET is a WSRESOURCE example. It is mapped to a set of orders

Example: WSRESOURCE-ORDERSET
has_name: order set
has_description: the wsresource representing order set
map: {onto:instance | isA(onto:instance, onto:ORDER)}

Table 6.5: RESTWS-ORDERSET is an example of Type I RESTful WS. RESTWS-ORDERSET is a RESTful WS supporting operations of GET, PUT, DELETE and POST on WSRESOURCE-ORDERSET

Example: RESTWS-ORDERSET
has_name: orderset-restws
has_description: RESTful WS serving the order set
has_URI: http://some.com/[customer_id]/orderset
has_type: type I
has_wsresource: onto:#WSRESOURCE-ORDERSET
onGET: onto:opn:Supported
onPUT: onto:opn:Supported
onDELETE: onto:opn:Supported
onPOST: onto:opn:Supported

Let's take the service serving the list of ORDERS as an example. The description of its corresponding WSRESOURCE and RESTWS is as follows:

Modeling Type II Web service Similar to defining Type I WSs, we define the WSRESOURCE first in the ontology, then associate the Web service with the defined WSRESOURCE. WSRESOURCE of type II Web service is mapped to a particular ontology instance.

Let's take the service providing ORDER information as an example. This WS supports three operations: (1) GET returns the detailed order representation, (2) PUT updates the order representation and (3) DELETE deletes the order representations. The description of its corresponding WSRESOURCE and RESTWS in Table 6.7 and Table 6.8:

Table 6.6: Type II RESTful Web Service

Def RESTWS II:	
has_name:	onto:restws#ws-name
has_description:	onto:restws#description
has_URI:	onto:restws#uri
has_type:	onto:restws#type
has_wsresource:	onto:wsresource
onGET:	onto:opn:Supported
onPUT:	onto:opn:Supported
onDELETE:	onto:opn:Supported
onPOST:	onto:opn:NotSupported

Table 6.7: WSRESOURCE-ORDER is another WSRESOURCE example. It is mapped to an individual order

Example: WSRESOURCE-ORDER
has_name: order-wsresource
has_description: the wsresource representing [order_id]
map: onto:#[order_id]

Table 6.8: RESTWS-ORDER is an example of Type II RESTful WS. POST is not supported by Type II RESTful WS.

Example: RESTWS-ORDER
has_name: order-restws
has_description: Web service serving order infor
has_URI: http://some.com/[customer_id]/[order_id]
has_type: type II
has_wsresource: onto:WSRESOURCE-ORDER
onGET: onto:opn:Supported
onPUT: onto:opn:Supported
onDELETE: onto:opn:Supported
onPOST: onto:opn:NotSupported

Modeling Type III Web service transitional RESTful WSs support only POST operation which causes "state transfer" between resources. In other words, the state of the resources will change themselves based on the request and state of other related resources, guided by certain rules. We use SWRL [41] rules to describe the functionality of this type of RESTful Web service. SubmitPayment and ShipOrder in the scenario(Figure 6.2) are of this type.

Def RESTWS III:	
has_name:	onto:restws#ws-name
has_description:	onto:restws#description
has_URI:	onto:restws#uri
has_type:	onto:restws#type
has_wsresource:	onto:set(onto:wsresource)
onGET:	onto:opn:NotSupported
onPUT:	onto:opn:NotSupported
onDELETE:	onto:opn:NotSupported
onPOST:	swrl:rule

For the instance of the service ShipOrder,

Example: WSRESOURCE-SHIPMENT

has_name: shipment-wsresource

has_description: wsresource [shippment_id]

map: onto:#[shippment_id]

Example: RESTWS-SHIPORDER

has_name: ShipOrder-restws

has_description: shipment service

has_URI: http://.../[shippment_id]/shiporder

has_type: type III

has_wsresource: {onto:WSRESOURCE-ORDER,

onto:WSRESOURCE-SHIPMENT}

onGET: onto:opn:NotSupported

onPUT: onto:opn:NotSupported

onDELETE: onto:opn:NotSupported

onPOST:

```
<ruleml:Imp>
 <ruleml:body rdf:parseType="Collection">
   <swrl:ClassAtom>
     <swrl:classPredicate
                  rdf:resource="#onto:Order"/>
     <swrl:argument1 rdf:resource="#o" />
   </swrl:ClassAtom>
   <swrl:IndividualPropertyAtom>
     <swrl:propertyPredicate
                 rdf:resource="#onto:isPaid"/>
     <swrl:argument1 rdf:resource="#o" />
   </swrl:IndividualPropertyAtom>
 <uleml:body>
 <ruleml:head rdf:parseType="Collection">
   <swrl:IndividualPropertyAtom>
     <swrl:propertyPredicate
               rdf:resource="#onto:isShipped"/>
     <swrl:argument1 rdf:resource="#o" />
   </swrl:IndividualPropertyAtom>
 </ruleml:head>
</ruleml:Imp>
```

The rule associated with action shipOrder is that: if the input is an order and the order has been paid, the state of the order should be updated as "isShipped" after this action is completed successfully.

6.3 Automating RESTful Web Service Composition

By establishing the model for describing RESTful WSs as ontology resources and "state transfer" of ontology resources, we form a conceptual model which can be used to facilitate automated composition of RESTful WSs. In this section, we present a situation calculus [52, 80] based sate transition system (STS) to automate the composition process of RESTful WSs.

6.3.1 Situation Calculus based State Transition System

Situation calculus is a first order logic based framework for representing changes and actions, and reasoning about them. It uses *situations* to represent the state of the world, and *fluents* to describe the changes from one situation to the other caused by the actions. We briefly explain the components of situation calculus:

• **Actions** are first order terms, $A(\vec{x})$, each consisting of an action name, A, and its argument(s), $\vec{x}$. In our STS, the actions are the HTTP operations (GET, PUT, POST and DELETE) applied to RESTful WSs. We use service name and HTTP operation (serviceName_operation) to represent the action name, and the argument(s) of the HTTP operations as the argument(s) of the action. We list the possible operations in the table below:

Actions:
TYPE I RESTWS + GET
TYPE I RESTWS + PUT
TYPE I RESTWS + POST
TYPE I RESTWS + DELETE
TYPE II RESTWS + GET
TYPE II RESTWS + PUT
TYPE II RESTWS + DELETE
TYPE III RESTWS+ POST

For example,

(1)The action of getting the order information is denoted as *RESTWS-ORDER_GET()*.

(2)The action of placing a new order is denoted as *RESTWS-ORDERSET_POST(o)*, where o is a variable denoting the resource representation of order.

(3)The action of submitting payment information to an order can be denoted as *RESTWS-SUBMITPAYMENT_POST(p, o)*, where p is a variable denoting the resource representation of payment.

(4)The action of cancel a payment to an order can be denoted as
RESTWS-PAYMENT_DELETE(p, o)

• A **situation** is a sequence of actions describing the state of the world and usually represented by symbol $do(a, s)$. For example,

(1)$do\ (RESTWS - ORDERSET\_POST(o), s_0)$ denotes the situation obtained after performing $RESTWS - ORDERSET\_POST(o)$) in the initial situation s_0 (s_0 is a special

situation which is not represented using a *do* function).

(2)$do\ (RESTWS - SHIPORDER\_POST(o),$
$do(RESTWS - SUBMITPAYMENT\_POST(p, o),$
$do(RESTWS - ORDERSET\_POST(o), s_0)))$ represents the situation obtained after performing the action sequence ($RESTWS - ORDERSET\_POST(o)$, $RESTWS - SUBMITPAYMENT\_POST(p, o)$, $RESTWS - SHIPORDER\_POST(o)$) in s_0.

• **Fluents** are situation-dependent relations and functions whose truth values vary from one situation to another. For example, $isPaid(o, s)$ represents if the payment has been submitted to the order denoted by o in situation s. $isShipped(o, s)$ means the order identified in o has been shipped in s. In our formal model,

$isPaid(o, s) \equiv onto : ORDER(o).isPaid$
$isShipped(o, s) \equiv onto : ORDER(o).isShipped$

• **Action precondition axioms:** For each action we may define one axiom, : $poss(a(\vec{x}, s) \equiv \Pi(\bar{(}x), s)$, which characterizes the precondition of the action. For example,
the precondition axiom of action $RESTWS - SHIPORDER\_POST(o)$ is:

$isPaid(o, s) \Rightarrow Poss(RESTWS - SHIPORDER\_POST(o), s)$

where $Poss$ denotes the possibility of performing the action.

• **Successor state axioms** are axioms that describe the effects of actions on fluents. Hence there is one such axiom for each fluent. For example, the successor state axiom for fluent $ReceiveOrder(o)$ is:

$Poss(a, s) \Rightarrow isPaid(o, do(a, s)) \Leftrightarrow a = RESTWS - SUBMITPAYMENT\_POST(p) \vee (isPaid(o, s) \wedge a \neq RESTWS - PAYMENT\_DELETE(p, o)$

In other words, the order is paid in the situation that results from performing the action if and only if we performed the $RESTWS - SUBMITPAYMENT\_POST(p)$ action or we already have it in the current situation and do not perform an action $RESTWS - PAYMENT\_DELETE(p, o)$ that will delete the payment.

• **Regression:** Regression is a mechanism for proving consequences in situation calculus. It is based on expressing a sentence containing the situation $do(a, s)$ in terms of a sentence containing the action a and the situation s, without the situation $do(a, s)$. The regression of a sentence φ through an action a is φ' that holds prior to a being performed iff φ holds after a. Successor state axioms support regression in a natural way [18]. Suppose that a fluent F's successor state axiom is $F(\vec{x}, do(a, s)) \Leftrightarrow \Phi_F(\vec{x}, a, s)$, we inductively define the regression of a sentence whose situation arguments all have the form $do(a, s)$:

$Regr(F(\vec{x}, do(a, s))) = \phi_F(\vec{x}, a, s)$

For other properties of regression, see [18].

$Regr(F(\neg\psi)) = \neg Regr(\psi)$
$Regr(F(\psi_1 \wedge \psi_2)) = Regr(\psi_1) \wedge Regr(\psi_1)$
$Regr(F(\exists x\psi)) = (\exists x)Regr(\psi)$

After representing the initial state s_0 and goal s using situation calculus, the composition problem is converted into a well-formed state transition problem. And the transition problem

can be solved with regression mentioned above, more specific details about solving a situation calculus based state transition problem can be seen in [52, 80].

Chapter 7

Related Work

In the past decade, much research effort [78] has been put on automated approaches to WSDL/SOAP Web service composition. However, automated RESTful Web service composition in the context of SOA is much less explored. Since WSDL/SOAP WSs and RESTful Web service adopt differing styles (imperative against declarative) and view the services from two different perspectives (operation-centric against resource-centric), the composition problem of these two kinds of WSs are very different.

WSDL/SOAP Web service composition predominately uses AI planning approaches, and these approaches focus on functional composition of individual WSs. That is, how to compose a new functionality out of existing component functionalities. However, RESTful WSs model the system from the perspective of resources. The composition of RESTful WSs focuses on the resource composition and "state transfer" between candidate WSs. In this Chapter, we survey the related work on both WSDL/SOAP based WSC problem and RESTful WSC problem.

7.1 Planning Based Web Service Composition

Several approaches have been proposed to address the WS composition problem with varying levels of automation. In this section, we survey existing planning approaches to automatic WSDL/SOAP WSC and briefly compare them with Haley. We position our work in the SOA context and discuss the limitations of previous research in this area.

SHOP2 [58], a classical planner based on hierarchical task networks, exploits the hierarchy for composing WSs as shown in [99, 86]. The authors define a translation from DAML-S(and

later OWL-S as well) process models to the SHOP2 domains, and from DMAL-S/OWL-S composition tasks to SHOP2 planning problems. They have implemented the translation and utilized an extended SHOP2 to do planning over the translated domain, and then executes the resulting plans. The final plan generated by the approach is a sequence of WS invocations and fails to account for uncertainties such as WS failures. Improving on this approach, Kutter et al. [45] attempt to deal with this issue by gathering information during planning, which may improve the robustness of the plans because information used to generate a plan may not change much at execution time. In comparison, Haley explicitly models uncertainty in WS outcomes and generates a policy which specifies a WS to invoke for every state of the composition.

In [53, 54], McIlraith et al. transform DAML-S based WS descriptions into situation calculus and implement the WS descriptions using Con-Golog interpreter [36], an extended version of Golog, that combines online execution of information-providing services with offline simulation of world altering services. They propose con-Golog language to enable programs that are generic, customizable and usable in the context of the Web. Theorem provers are used to arrive at a plan which is a sequence of WS invocations. We improve on this work by modeling the uncertain behavior of WSs, first by using a probabilistic variant of situation calculus and second, by using decision-theoretic planners. Therefore, we offer a way to form WS compositions that are more robust to WS failures and other events.

Medjahed et al. [56, 55] present a technique to generate composite services from high-level declarative descriptions of the individual services. The method uses compensability rules, defining possible WS attributes that could be used in service composition, to determine whether two services are composable. It provides a way to choose a plan in the selection phase based on the quality of composition parameters (e.g. rank, cost, etc.). But the final plan is not a conditional plan; it may fail to adjust properly to the dynamic changes in the environment.

Traverso and Pistore [88, 72] propose a MBP (a model checking planner) based framework to automate WS composition, where WSs are modeled as having stateful, non-deterministic and partially observable behaviors. Our approach improves on this line of work in that Haley not only handles the non-determinism of WSs, but offers a way to scale WS composition using a hierarchical approach. Pistore et al. [73] improving on their previous work [88] transform composite WSs described using BPEL4WS into a KB and apply MBP to arrive at a plan for composing the WSs. While MBP handles non-determinism, the language used for the KB is restrictive and the final plan does not provide cost guarantees. In addition to handling uncertainty and providing cost-based optimality, Haley is scalable and allows the full generality of first order logic based descriptions of WSs.

Oh, Lee and Kumara proposed a forward and backward (bidirectional) search based approach [67, 68] for WSC. They compared their algorithm with other approaches in two simplistic WSC benchmarks as part of the WS Challenge [1], exhibiting good results in terms of the speed of composition. Plans produced in this approach are based on the reachability analysis of input and output variables only. In other words, only the input and output as the functional description of WSs is considered. This may not be realistic because multiple WSs with the same input and output often exist and the approach provides no way to choose between them. In particular, non-functional parameters of WSs are ignored, which typically allows the selection of a composition among many candidate ones.

Recently, Qiu et al. [76, 77, 75] proposed a context optimization and planning based approach for semantic WSC. A context-aware planning method comprising of global planning and local optimization based on context information is utilized. In particular, a framework is introduced for composing semantic WSs using backward chaining based search to find candidate services via reasoning in description logics. This is combined with a DAG-based method to select services and formulate the planning problem, and filtering of inappropriate services during the DAG generation. Although context is incorporated into planning, this

[1] WS Challenge: http://www.ws-challenge.org/

approach does not model the uncertain behaviors of WSs. Furthermore, it does not select WSs based on QoS or performs process optimization.

In SELF-SERV [84, 14], web services are declaratively composed and then executed in a dynamic peer-to-peer environment. SELF-SERV compose WSs based on state-charts, gluing together an operation's input and output parameters, consumed and produced events. Service execution is monitored by software components called coordinators, which initiate, control, and monitor the state of a composite service they are associated with. The coordinators retrieve the state relevant information from the service's state-chart and represent it in what is called a routing table containing pre-conditions and post-processions. This system provides tools for specifying composite services, data conversion rules, and provider selection policies. These specifications are then translated into XML documents that can be interpreted by peer-to-peer inter-connected software components to provision the composite service without requiring a central authority. ELF-SERV may be classified to be a toolkit for manually composing WSs rather than an automated Web service composition approach.

Wu et al. [100] presents an automatic approach for Web service composition, while addressing the problem of process heterogeneities and data heterogeneities by using an extended GraphPlan planner and a data mediator. An extended GraphPlan algorithm is employed to generate a BPEL process based on the task specification and candidate Web services described in SAWSDL. In addition, the authors address the problem of structural heterogeneities in message schema by having the developer associate mappings using the Schema Mapping on Web service message (input and output) elements.

Another closely related research topic is automatic workflow composition.

Chun et al. [27] presents a formal model for automatic workflow generation that uses domain knowledge represented as service ontology, regulatory ontology and a user profile. Compositional rules consist of selection rules that select obligatory and preference tasks, and coordination rules that glue tasks together in order. Each rule is represented as Condition-

Action pair. The composition algorithm evaluates compositional rules against a user profile, automatically generating the customized inter-agency workflow.

Korhonen et al. [44] presents a way to use ontology-based reasoning to automatically combine component workflow instances. The web services workflows are described using a transactional workflow ontology. The workflow ontology can be used to describe both component web service workflows and master web service workflows. The authors have also implemented a workflow engine that runs the workflow instances. A reasoning agent is utilized to automatically find a composed workflow that fulfills all given constraints. The result from the inference is a workflow instance that can be executed using the implemented workflow engine.

Albert, Henocque and Kleiner [12] presents a constrained object model for workflow composition, based upon a metamodel for workflows and ontologies for processes and data flows. This research shows the feasibility of using configuration techniques to achieve automatic workflow composition. This possibility acknowledges the fact that workflow composition can be seen as a finite model search problem.

Lu, Bernstein and Lewis [47] extends the semantic correctness theory to modeling and reasoning about workflows. Specifically, the aurhors develop a formal model in which workflows are assumed to be constructed from a library of tasks to promote task reuse. The semantics of tasks and workflows is specified in terms of pre- and postconditions, and a sound inference rule is provided to precisely specify each of our workflow constructs. Based on this model we develop algorithms that automatically: (1) verify if a workflow implementation satisfies its specification, (2) synthesize a workflow implementation from the workflow description and a given task-library.

These work [12, 27, 44, 47] on automatic workflow composition focus on the functionality synthesis. Many of these approaches propose their own ontologies to describe service components and rely on the ontology inference to generate a workflow satisfying the given functional requirements. In contrast, our approach utilizes commonly-accepted WS descrip-

tion standards and relies on decision-theoretic planning to generate a composition satisfying the goals, while taking into account quality measurements such as response time, reliability and invocation costs.

7.2 Configuration Based Web Service Composition

METEOR-S [11, 22, 23] aims to support the complete life cycle of semantic Web processes. At the composition stage, it manually configures the process as a "Semantic Process Template" and dynamically chooses the candidate WSs for the abstract components in the process. It presents a constraint driven WS composition tool, which allows the process designers to bind WSs to an abstract process, based on business and process constraints.

Similar to METEOR-S, Zeng et al. [101, 102, 103] proposes a global planning approach to optimally select component services during the execution of a composite service. Service selection is formulated as an optimization problem which can be solved using efficient linear programming methods. By sharing the similar view of selecting services as an optimization problem, Canfora and Esposito [21] proposed a lightweight approach for QoS aware service composition using genetic algorithms. Also, the paper presents an algorithm for early triggering service re-planning.

As pointed by Wiesemann et al. [96], many optimization-based approaches to the service composition problem treat the QoS of a service as deterministic quantities. As a contrast, Wiesemann et al. view these QoS parameters as stochastic variables quantified with average value-at-risk (AVaR). It formulates the service selecting problem as a multi-objective stochastic program which simultaneously optimizes QoS parameters: duration, service invocation costs, availability, and reliability. The model minimizes the average value-atrisk (AVaR) of the workflow duration and costs while imposing constraints on the workflow availability and reliability.

Compared to planning based approaches, these approaches [11, 101, 21, 96] do not automatically compose individual WSs into processes, but focuses on the dynamic selection of

candidate WSs for the functional components in the process. The process is assumed to have been manually configured beforehand.

7.3 Web Service Composition Tool Support

While a variety of WSC algorithms and approaches have been proposed, few implemented tools or solutions are available to support automated WSC. This is because of the complex nature of the WSC problem and the inherent scalability issues in existing AI planners. Two exceptions are Synthy [10, 24] and the Astro Project [2] .

Synthy accepts WSs described in OWL-S and composes the WSs in two stages: The first stage composes an abstract workflow to satisfy the functional requirements, and the second stage chooses WS instances for the components in the abstract workflow, based on the QoS attributes of WS instances. Although Synthy utilizes QoS properties of WSs, it, however, does not specify how these QoS properties are specified or acquired.

The Astro tool suite offers a way to compose WSs described using abstract BPEL into business processes. It supports both WSC design and execution. Using abstract BPEL to describe WSs is an unintuitive choice given the available spectrum of WS description languages. In particular, designing WSs using abstract BPEL is itself a time-consuming and cumbersome process. While both Synthy and Astro utilize planning, neither of them address the scalability issues of AI planning algorithms in any feasible way. This could affect the adoption of these two tool suites and their use in large business process composition scenarios.

To the best of our knowledge, there is no previous attempts towards a formal modeling of RESTful WSs in terms of facilitating automated service composition. The following two topics are loosely related to our presented work, although none of these discusses the specific issue of automated RESTful Web service composition.

[2]Astro Project: http://www.astroproject.org/

7.4 RESTful Web Service Description Languages

An increasing number of leading Internet companies (such as Google, Yahoo, Amazon and etc.) are developing REST style services that provide API access to their internal data and resources. Unlike WSDL/SOAP WSs, these applications are typically described using informal textual documentation (HTML, PDF, DOC and etc.) supplemented with more formal specifications such as XML schema for data formats. Although the Web service community is still debating if RESTful WSs really need a formal description language due to its simplicity and self-declarative nature, we do need a formal, machine-understandable description language to enable automated RESTful Web service composition. Indeed, we might not need a formal description document like WSDL to write a client program consuming RESTful WSs, but if we seek to automate the process of composing RESTful Web service from the pool of vast amount of candidate RESTful WSs, a formal machine-understandable description model is needed.

Web Application Description Language (WADL) [38] is designed to provide a machine process-able description of REST style Web applications. It is targeted to provide a simple alternative to WSDL for use with REST style Web services. Analogous to WSDL2Java for WSDL, WADL provides a tool called WADL2JAVA [37] that that generates client side stubs from WADL files. The focus of WADL is to describe the data format of request parameters and response messages. WADL defines both resources and representations, as well as the HTTP methods that can be used to manipulate the resources. "Resources" in WADL are abstracts of services, and not associated with domain resources.

Semantic Annotations for REST (SA-REST) [46, 85] is a research effort to semantically describe RESTful WSs by adding annotations to HTML pages that describe RESTful WSs. SA-REST is inspired by the idea of grounding service descriptions to semantic meta models via model reference annotations from SAWSDL. However, unlike formal description languages like WSDL for traditional WSs, RESTful WSs are usually described by informal documents like HTML web pages. Consequently, SA-REST uses Resource Descrip-

tion Framework-in-attributes (RDFa) [95] and Gleaning Resource Descriptions from Dialects of Languages (GRDDL) [94] to add semantic annotations.

Web Services Description Language (WSDL) Version 2.0 [26, 49] has enhanced its support for HTTP bindings to describe REST style WSs. Unlike WADL or SA-REST, WSDL 2.0 is the second version of WSDL, of which the original purpose is to describe traditional operation-oriented style WSs. HTTP operations in REST is supported in WSDL 2.0 by adding HTTP bindings. All 4 HTTP operations are currently supported in WSDL 2.0 along with other RPC style operations, as well as the input parameters and response messages are described in the same fashion as the counterparts of operation-oriented operations.

These languages are strongly influenced by existing imperative service description languages and do not capture well the resource-centric nature of RESTful WSs. They have focused on the descriptions of input/output as traditional service description languages do, but ignored the description of the resources and the transitions of these resources. As a consequence, these languages remain at the interface description level and are not capable of capturing the "state transfer" between resources. Thus, they can not be directly used to facilitate automated composition of individual RESTful WSs.

7.5 Mashup

Mashup is a Web application that combines data from multiple data sources into a single integrated application. A mashup site must access third party data using APIs, and should add value to these data during the integration. Data could come from local databases or various sources across the Internet via different protocols including RSS [98], ATOM [59] and RESTful WSs. Compared to RESTful Web service composition, a mashup is restricted at the data-level integration, and most uses of RESTful WSs in mashup are limited to fetching data from remote sources. It usually does not involve updating or manipulating remote data sources or other resources.

Chapter 8

Conclusions

In this chapter, we summarize the challenges and our contributions to the problem of automated WS composition and discuss some open issues and future research directions for WS composition.

8.1 Summary and Discussion

In terms of WSDL/SOAP WS composition, this article introduced a hierarchical symbolic decision-theoretic planning based approach for composing WSs. We focused on addressing three key challenges faced by contemporary approaches, which make WSC a difficult problem: (1) Non-deterministic WS behaviors; (2) the benefits of optimality of the WSC; and (3) scalability of the compositions. Haley utilizes a stochastic planner for the composition, thereby offering a natural way to handle the uncertainty associated with WSs. The composition process takes into account both functional and non-functional parameters (response time, invocation cost and reliability) and provides a cost based WSC optimization. It addresses the scalability issue by adopting a symbolic representation and utilizing a hierarchical approach. Specifically, symbolic representation helps us address two primary issues: First, is the explosion in the state space as the number of WSs increase impacting the scalability of the composition methods. Second, is the capability to operate directly on WS descriptions – WS preconditions and effects which may be represented using first order logic based languages. This enables Haley to directly elicit a planning problem formulation from service descriptions. Additionally, we observe that many real world business processes are amenable to a hierarchical decomposition into lower level processes and primitive service invocations. Haley

utilizes a framework that models the hierarchy often found in processes. Our experimental evaluation shows that the framework outperforms other approaches in dynamic, stochastic environments, and is efficient and scalable.

In addition to contributing a theoretical framework for composing WSs, we have implemented it as a working system and provided a set of supporting tools. Haley is available as a state of the art, end-to-end and scalable solution for WSC. It provides an interface that hides the complexity of planning and WS-BPEL from process designers. The tool suite offers unique advantages over manual BPEL process design and other automated approaches to composition. Development of Haley represents a significant milestone in the research on WSC, and a contribution to application development in SOA.

An associated challenge is the data mediation problem, which is not addressed in this dissertation. Specifically, both syntactical and semantic heterogeneity may exist in the input and output messages exchanged between WSs. In other words, the output of the previous WSs may not exactly match the required input of the successive WSs. Data mediation provides a formalized model and mechanism for managing data heterogeneity which may exist in the component WSs. It is a challenging problem and is beginning to receive renewed research attention in the semantic Web service community. One proposed approach [57] models the involved domains using ontologies and relies on the pre-constructed data mappings to solve the heterogeneity issue. Our framework could be extended in a straightforward manner with approaches that address the data mediation challenges.

Due to the declarative nature and other characteristics of RESTful WSs such as being light-weight, easily accessible and scalable, we argued that RESTful WSs have some unique advantages over traditional WSs in terms of service composition, especially in the context of building Web 2.0 applications. While RESTful WSs have been widely used in building mashup applications, we believe RESTful WSs will be playing an increasingly important role in the context of SOA, where WSDL/SOAP WSs are dominant.

The RESTful approach represents a very promising way of building WSs. Although it has been considered as an important technology to realize programmable Web in the industry, and potentially adopted as widely as WSDL/SOAP Web service composition, we did not see research effort towards RESTful Web service composition because it is a relatively new technology. In this dissertation, we discuss two perspectives of modeling a system using WSs. We introduce a formal conceptual model for describing individual RESTful WSs (identified as three types), and present an automated composition framework based on this model. This work represents our initial efforts towards the problem of automated RESTful Web service composition. We are hoping that it will draw some interests from the research community on WSs, and engage more researchers in this challenge.

We outline below the challenges we encountered towards automated RESTful Web service composition.

- Resource-centric perspective of building services is relatively new, and most of the claimed RESTful WSs do not fully adhere to REST principles.
- Lack of formal modeling or machine-understandable description languages for RESTful WSs.
- While integrating RESTful WSs (resources) from multiple parties, data heterogeneity may become a major obstacle.

8.2 Original Contributions and Significance

Our research focuses on two closely related but fundamentally different topics: automated WSDL/SOAP WS composition and RESTful WS composition. Automated WSDL/SOAP WS composition problem has been extensively studied in the WS research community, and many approaches have been proposed. We start our research by analyzing the achievements and limitations of existing approaches. We have identified some of the key issues, such as uncertainty, optimality, scalability and etc., needed to be solved by WSC approaches. Our

research on WSDL/SOAP WS composition focuses on addressing these key issues. In addition, our research on automated RESTful WS composition represents early research effort on this topic in the research community. We propose a classification based formal ontology model to describe individual WSs, and a state transition system based framework to automatically compose RESTful Web services. We summarize our contribution below

- In order to model uncertainties during WS invocation and optimize QoS requirements, we introduce the use Decision-theoretic planning in WS composition. Decision-theoretic planning generalize classical planning techniques to nondeterministic environments where action outcomes may be uncertain, and associate costs to the different plans thereby allowing the selection of an optimal plan. Compared to classical AI planners bases approaches, our approach models the uncertainty inherent in WSs and facilitates a cost-based process optimization. These techniques are especially relevant in the context of SOAs where services may fail and processes must minimize costs.

- Many real world processes are amenable to a hierarchical decomposition into lower level processes and primitive service invocations. We present a hierarchical approach for composing processes that may be nested some of the components of the process may be sub-processes themselves. In particular, the lowest levels of the hierarchy (leaves) are modeled using a FO-SMDP containing *primitive* actions which are invocations of the WSs. Higher levels of the process are modeled using FO-SMDPs that contain *abstract* actions as well, which represent the execution of lower level processes. We represent their invocations as *temporally extended* actions in the higher level FO-SMDPs. Since descriptions of only the individual WSs are usually available, we provides methods for deriving the model parameters of the higher level FO-SMDP from the parameters of the lower level ones. Thus, our approach is applicable to WSCs that are nested to an arbitrary depth. The method of constructing the primitive SMDP planning problem from Web service composition problem is proposed, and more importantly we provide ways for deriving the parameters of the composite SMDPs from the lower level ones.

- Although our hierarchical approach promotes scalability, along with existing WSC techniques, it is further plagued by two challenges: (i) As the number of WSs increases, there is an explosion in the size of the state space representation; (ii) There is a growing consensus among the WS description standards such as OWL-S [50] and SAWSDL on using first order logic (or its variants) to logically represent the preconditions and effects of WSs. However, many of the existing planning techniques used for WS composition do not use the full generality of first order logic while planning. We propose a first-order hierarchical decision-theoretic planning framework, which we call Haley [105], for WSC problems. Haley improves on previous work by allowing WS composition at the logical level. Specifically, Haley enables composition using the first order sentences that represent the preconditions and effects of the component WSs. Haley offers a way to mitigate the problem of large state spaces by composing at the logic level and preserves the expressiveness of first order logic. Haley operates directly on first order logic based representations of the state space to obtain the compositions. As a result, it supports an automated elicitation of the corresponding planning domain from Web service descriptions and produces a much more compact domain representation than classical AI planners.

- Although many approaches have been proposed in the literature, few implemented tools exist due to the limitations of the existing approaches mentioned above and the complexity of the WS composition problem itself. We have implemented Haley and provided a comprehensive tool suite. The suite accepts WSs described using standard languages such as SAWSDL. It provides process designers with an intuitive interface to specify process requirements, goals and a hierarchical decomposition, and automatically generates executable BPEL processes, while hiding the complexity of the planning and BPEL from users. At the core of Haley, we have designed a first-order SMDP based decision-theoretic planner, eDT-GOLOG, which may be used independent with Haley. eDT-GOLOG is used as the planning engine in Haley to generate the plan based

on WS descriptions and user-specified goals. We also designed a SAWSDL viewer to visualize the functionalities of WSs. In a nutshell, the tool suite is an integrated development environment for the process designers to (1) import candidate WSs and their description files, (2) specify process hierarchies, initial state and goals, (3) generate the planning domain, (3)generate the plan and (4) convert the plan into the corresponding BPEL file.

- Our Experiments [104, 105] demonstrate that the advantages of a hierarchical decomposition and logic based representation. The hierarchical approach consumes less planning time than the flat approach; and the first order representation produces more compact planning domains.

- To the best of our knowledge, automated RESTful WS composition is much less studied in the WS research community than WSDL/SOAP WS composition. We have put our initial efforts into RESTful WS modeling and composing. We are hoping our research will attract more interests and engage more researches from the WS research community. As an early research effort towards automated RESTful WS composition, we introduce and formally define this problem. In addition, we have analyzed the differences between automated WSDL/SOAP WS composition and automated RESTful WS composition. We have identified a list of challenges of addressing automated RESTful WS composition. To facilitate automated RESTful WS composition, we have proposed a classification based formal modeling method to describe RESTful WSs. A situation calculus based state transition system has been studied to automate the composition of RESTful WSs.

8.3 Future Work

There are several research directions to further improve or extend our dissertation work. We divide these directions into two categories: future work of WSDL/SOAP WS composition and future work of RESTful WS composition.

Future work of WSDL/SOAP WS composition: We have addressed some of the key issues in our dissertation work, but there are still some open issues and unsolved problems. To improve Haley, one may consider the following directions:

(1) Data mediation has not been addressed in the Haley framework. Specifically, both syntactical and semantic heterogeneity may exist in the input and output messages exchanged between WSs. In other words, the output of the previous WSs may not exactly match the required input of the successive WSs. Data mediation is a challenging problem and is beginning to receive renewed research attention in the semantic Web service community. One may extend our framework by providing a sf Haley plug-in to address the data mediation challenges.

(2) While three major quantitative QoS parameters (Response time, reliability and invocation cost) are considered in Haley, other QoS parameters (qualitative QoS parameters) such as compliance, security, etc. have not been addressed in our approach. These unincorporated QoS measurements might be crucial in certain application domains. Since Haley utilizes a cost-based planning engine, one might quantify these parameters in order to take into account these qualitative QoS parameters. The three quantitative QoS parameters currently considered in Haley are combined in a sensible way to produce a single optimization goal. To incorporate more QoS parameters, one needs to carefully analyze the connections among the newly included parameters and the old parameters already considered.

(3) Our current approach for optimizing multiple QoS parameters is to combine these parameters into a single optimization goal, as a consequence, a single optimal policy is produced. It is also possible to view the problem as a multiple-objective optimization problem.

For instance, one might apply the "pareto optimization" idea to Haley and produce a "pareto policy set", rather than a single policy.

(4) As far as the tool suite is concerned, Haley may be tested with additional real world large-scale scenarios and continue to be improved on the usability and reliability of the tool suite.

Future work of RESTful WS composition: This topic is relatively new to the researchers in the WS community. Much less research has been done on this problem. Our work is also still preliminary and may be improved in various ways.

(1) Our current work focuses on the service composition approach and leaves the description of RESTful WSs at a conceptual level, although we have proposed an ontology based model to describe RESTful WSs and facilitate automated RESTful WS composition. Most uses of RESTful WSs in the industry follow an ad-hoc approach by looking at the informal service description document. The informal nature of these documents is a big obstacle to applying automated service compositions. To further realize the potential of RESTful WSs, especially in terms of facilitating automated composition, a formal, machine-understandable description language is needed.

(2) RESTful WSs and WSDL/SOAP WSs have shown advantages and disadvantages in different application situations. It would be interesting to study how we can automate the service composition process in an environment mixed with these two WS paradigms.

Bibliography

[1] Amazon web services. http://aws.amazon.com/.

[2] Eclipse graphical modeling framework (GMF). http://eclipse.org/modeling/gmf/.

[3] Eclipse modeling framework project (EMF). http://eclipse.org/modeling/emf/.

[4] Organization for the advancement of structured information standards (oasis). http://www.oasis-open.org.

[5] Usps web services. http://www.usps.com/webtools/.

[6] Webservicex. http://www.webservicex.net/WCF/webServices.aspx.

[7] World wide web consortium (w3c). http://www.w3.org/.

[8] Prgrammable web - api dashboard. http://www.programmableweb.com/apis, 2009.

[9] (ws-*) list of web service specifications. http://en.wikipedia.org/wiki/WS-*, 2009.

[10] V. Agarwal, G. Chafle, K. Dasgupta, N. M. Karnik, A. Kumar, S. Mittal, and B. Srivastava. Synthy: A system for end to end composition of web services. *Journal of Web Semantics*, 3(4):311–339, 2005.

[11] R. Aggarwal, K. Verma, J. A. Miller, and W. Milnor. Constraint driven web service composition in meteor-s. In *Proceedings of IEEE International Conference on Services Computing*, pages 23–30, 2004.

[12] P. Albert, L. Henocque, and M. Kleiner. A constrained object model for configuration based workflow composition. In *Business Process Management Workshops*, pages 102–115, 2005.

[13] R. E. Bellman. *Dynamic Programming*. Dover, 1957.

[14] B. Benatallah, Q. Z. Sheng, and M. Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1):40–48, 2003.

[15] J. Blythe. Decision-theoretic planning. *AI Magazine*, 20(2).

[16] J. Blythe. Decision-theoretic planning. *AI Magazine*, 20(2):37–54, 1999.

[17] J. Blythe. An overview of planning under certainty. pages 85–110, 1999.

[18] C. Boutilier, R. Reiter, and B. Price. Symbolic dynamic programming for first-order mdps. In *IJCAI*, pages 690–700, 2001.

[19] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence*, pages 355–362, 2000.

[20] T. Bylander. Complexity results for planning. In *International Joint Conference of Artificial Intelligence*, pages 274–279, 1991.

[21] G. Canfora and R. Esposito. A lightweight approach for qos-aware service composition. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 36–47, 2004.

[22] J. Cardoso and A. P. Sheth. Semantic e-workflow composition. *Journal of Intelligent Information System*, 21(3):191–225, 2003.

[23] J. Cardoso, A. P. Sheth, J. A. Miller, J. Arnold, and K. Kochut. Quality of service for workflows and web service processes. *Journal of Web Semantics*, 1(3):281–308, 2004.

[24] G. Chafle, G. Das, K. Dasgupta, A. Kumar, S. Mittal, S. Mukherjea, and B. Srivastava. An integrated development environment for web service composition. In *IEEE International Conference on Web Services (ICWS)*, pages 839–847, 2007.

[25] R. Chinnici, J. J. Moreau, A. Ryman, and S. Weerawarana. Web services description language specification 1.1. http://www.w3.org/TR/wsdl, 2001.

[26] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language specification 2.0. http://www.w3.org/TR/wsdl20/, 2007.

[27] S. A. Chun, V. Atluri, and N. R. Adam. Domain knowledge-based automatic workflow generation. In *DEXA*, pages 81–92, 2002.

[28] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.

[29] T. D. S. Coalition. Daml-s specification. http://www.daml.org/services/daml-s/0.9/, 2003.

[30] T. M. K. Consortium. Kerberos: The network authentication protocol. http://web.mit.edu/Kerberos/, Feb. 2009.

[31] P. Doshi, R. Goodwin, R. Akkiraju, and K. Verma. Dynamic workflow composition: Using markov decision processes. *Journal of Web Service Research*, 2(1):1–17, 2005.

[32] J. Farrell, H. Lausen, and etc. Sawsdl: Semantic annotations for wsdl, 2006.

[33] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architecture*. PhD thesis, University of California, Irvine, 2000.

[34] R. T. Fielding. A little rest and relaxation. In *The International Conference on Java Technology (JAZOON)*, June 2007.

[35] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. In *ICSE*, pages 407–416, 2000.

[36] G. D. Giacomo, Y. Lespérance, and H. J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artif. Intell.*, 121(1-2):109–169, 2000.

[37] M. J. Hadley. Web application description language (wadl) official site. https://wadl.dev.java.net, 2006.

[38] M. J. Hadley. Web application description language (wadl) specification. https://wadl.dev.java.net/wadl20061109.pdf, 2006.

[39] D. Hirtle, H. Boley, B. Grosof, M. Kifer, M. Sintek, S. Tabet, and G. Wagner. Schema specification of ruleml, 2006.

[40] S. Holldobler and O. Skvortsova. A logic-based approach to dynamic programming. In *In Learning and Planning in Markov Processes-Advances and Challenges-AAAI 04 Workshop*, 2004.

[41] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. Swrl: A semantic web rule language combining owl and ruleml. http://www.w3.org/Submission/SWRL/, 2004.

[42] IETF. Public-key infrastructure (x.509). http://www.ietf.org/html.charters/pkix-charter.html, Aug. 2008.

[43] K. Kersting, M. V. Otterlo, and L. D. Raedt. Bellman goes relational. In *Proceedings of the twenty-first international conference on Machine learning (ICML)*, volume 69, page 59, 2004.

[44] J. Korhonen, L. Pajunen, and J. Puustjärvi. Automatic composition of web service workflows using a semantic agent. In *Web Intelligence*, pages 566–569, 2003.

[45] U. Kuter, E. Sirin, D. Nau, B. Parsia, and J. Hendler. Information gathering during planning for web serivce composition. *Journal of Web Semantics*, 3:183–205, 2005.

[46] J. Lathem, K. Gomadam, and A. P. Sheth. Sa-rest and (s)mashups : Adding semantics to restful services. In *Proceedings of the First IEEE International Conference on Semantic Computing*, pages 469–476, 2007.

[47] S. Lu, A. J. Bernstein, and P. M. Lewis. Automatic workflow verification and generation. *Theor. Comput. Sci.*, 353(1-3):71–92, 2006.

[48] H. Ludwig, A. Keller, A. Dan, R. P. King, and R. Franck. *Web Service Level Agreeement Lang. Spec.* 2003.

[49] L. Mandel. Describe rest web services with wsdl 2.0. Describe REST Web services with WSDL 2.0, 2008.

[50] D. Martin, M. Burstein, J. Hobbs, and etc. *OWL-S: Semantic Markup for Web Services.* 2006.

[51] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, and etc. Bringing semantics to web services: The owl-s approach. In *Proceedings of SWSWPC*, pages 26–42, San Diego, California, USA, July 2004.

[52] J. McCarthy. Situations, actions and causal laws. Technical report, AI Laboratory, Stanford University, 1963.

[53] S. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In *Proceedings of International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, pages 482–496, Toulouse, France, 2002.

[54] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic web services. 16:45–53, 2001.

[55] B. Medjahed and A. Bouguettaya. A multilevel composability model for semantic web services. *IEEE Transactions on Knowledge Data Engineering*, 17(7):945–968, 2005.

[56] B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing web services on the semantic web. *VLDB Journal*, pages 333–351, 2003.

[57] M. Nagarajan, K. Verma, A. P. Sheth, and J. A. Miller. Ontology driven data mediation in web services. *International Journal Web Service Research*, 4(4):104–126, 2007.

[58] D. S. Nau, T.-C. Au, O. Ilghami, and etc. Shop2: An htn planning system. *Journal of Artificial Intelligence Research (JAIR)*, 20:379–404, 2003.

[59] M. Nottingham and R. Sayre. The atom syndication format. http://www.atompub.org/2005/07/11/draft-ietf-atompub-format-10.html, 2006.

[60] OASIS. Uddi specification. www.oasis-open.org/committees/uddi-spec/.

[61] OASIS. Web services reliable messaging, ws-reliability 1.1. Web Services Reliable Messaging TC WS-Reliability 1.1, Nov. 2004.

[62] OASIS. Saml specification. http://saml.xml.org/saml-specifications, Mar. 2005.

[63] OASIS. Web services security: Soap message security 1.1. http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf, Feb. 2006.

[64] OASIS. Web services business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 2007.

[65] OASIS. Ws-trust 1.3. http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.html, Mar. 2007.

[66] OASIS. Web services atomic transaction (ws-atomictransaction). http://docs.oasis-open.org/ws-tx/wsat/2006/06, 2009.

[67] S.-C. Oh, H. Kil, D. Lee, and S. R. T. Kumara. Algorithms for web services discovery and composition based on syntactic and semantic service descriptions. In *The Third IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (EEE 2006)*, page 66, June 2006.

[68] S.-C. Oh, D. Lee, and S. R. T. Kumara. Web service planner (wspr): An effective and scalable web service composition algorithm. *Int. J. Web Service Res.*, 4:1–22, 2007.

[69] T. O'Reilly. What is web 2.0. http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html, 2005.

[70] T. O'Reilly. Web 2.0 compact definition: Trying again. http://radar.oreilly.com/archives/2006/12/web-20-compact.html, 2006.

[71] C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs. "big" web services: Making the right architecture decision. In *The Proceedings of International World Wide Web Conference (WWW)*, Apr. 2008.

[72] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. Planning and monitoring web service composition. In *Artificial Intelligence: Methodology, Systems, and Applications, 11th International Conference, AIMSA*, pages 106–115, Sept. 2004.

[73] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated composition of web services by planning at the knowledge level. In *IJCAI*, pages 1252–1259, 2005.

[74] M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience, 1994.

[75] L. Qiu, L. Chang, F. Lin, and Z. Shi. Context optimization of ai planning for semantic web services composition. *Service Oriented Computing and Applications*, 1(2):117–128, 2007.

[76] L. Qiu and Z. Shi. Context-aware services composition based on ai planning. In *INFORMATIK 2006*, pages 345–352, Oct. 2006.

[77] L. Qiu, Z. Shi, and F. Lin. Context optimization of ai planning for services composition. In *2006 IEEE International Conference on e-Business Engineering (ICEBE 2006)*, pages 610–617, Oct. 2006.

[78] J. Rao and X. Su. A survey of automated web service composition methods. In *Semantic Web Services and Web Process Composition*, pages 43–54, 2004.

[79] R. Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. *Artificial intelligence and mathematical theory of computation: papers in honour of John McCarthy*, pages 359–380, 1991.

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

[81] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly Media, Inc., 2007.

[82] S. Rusell and P. Norvig. *Artificial Intelligence, A Modern Approach*. Prentice Hall, 2nd edition edition, 2003.

[83] S. Sanner and C. Boutilier. Approximate linear programming for first-order mdps. In *Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence*, pages 509–517, 2005.

[84] Q. Z. Sheng, B. Benatallah, M. Dumas, and E. O.-Y. Mak. Self-serv: A platform for rapid composition of web services in a peer-to-peer environment. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 1050–1054, Aug. 2002.

[85] A. P. Sheth, K. Gomadam, and A. Ranabahu. Semantics enhanced services: Meteor-s, sawsdl and sa-rest. *IEEE Data Eng. Bull.*, 31(3):8–12, 2008.

[86] E. Sirin, B. Parsia, D. Wu, J. A. Hendler, and D. S. Nau. Htn planning for web service composition using shop2. *Journal of Web Semantics*, 1(4):377–396, 2004.

[87] M. Soutchanski. *High-Level Robot Programming in Dynamic and Incompletely Known Environments*. PhD thesis, University of Toronto, 2003.

[88] P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. In *International Semantic Web Conference*, pages 380–394, 2004.

[89] A. Turing. On computable numbers, with an application to the entscheidungs problem. *Procedings of the London Mathematical Society*, (42):230–265, 1936.

[90] W3C. Soap version 1.2 specification. http://www.w3.org/TR/soap12.

[91] W3C. Hypertext transfer protocol – http/1.0. http://www.w3.org/Protocols/HTTP/1.0/draft-ietf-http-spec.html, 1996.

[92] W3C. Cool uris don't change. http://www.w3.org/Provider/Style/URI, 1998.

[93] W3C. Xml schema. http://www.xml.org/xmlschema, 2005.

[94] W3C. Gleaning resource descriptions from dialects of languages (grddl). http://www.w3.org/TR/grddl/, 2007.

[95] W3C. Rdfa in xhtml: Syntax and processing. http://www.w3.org/TR/rdfa-syntax/, 2008.

[96] W. Wiesemann, R. Hochreiter, and D. Kuhn. A stochastic programming approach for qos-aware service composition. In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 226–233, 2008.

[97] E. Wilde07. Declarative web 2.0. In *Proceedings of the IEEE International Conference on Information Reuse and Integration*, pages 612–617, 2007.

[98] D. Winer. Rss 2.0 specification. http://validator.w3.org/feed/docs/rss2.html, 2002.

[99] D. Wu, B. Parsia, E. Sirin, and etc. Automating daml-s web services composition using shop2. In *Proceedings of International Semantic Web Conference*, pages 195–210, 2003.

[100] Z. Wu, K. Gomadam, A. Ranabahu, A. P. Sheth, and J. A. Miller. Automatic composition of semantic web services using process mediation. In *Proceedings of the Ninth International Conference on Enterprise Information Systems*, pages 453–462, 2007.

[101] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality driven web services composition. In *Proceedings of International World Wide Web Conference (WWW)*, pages 411–421, 2003.

[102] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.

[103] L. Zeng, A. H. H. Ngu, B. Benatallah, R. M. Podorozhny, and H. Lei. Dynamic composition and optimization of web services. *Distributed and Parallel Databases*, 24(1-3):45–72, 2008.

[104] H. Zhao and P. Doshi. A hierarchical framework for composing nested web processes. In *International Conference of Service Oriented Computing*, pages 116–128, 2006.

[105] H. Zhao and P. Doshi. Haley: A hierarchical framework for logical composition of web services. In *Proceedings of International Conference on Web Services*, pages 312–319, 2007.

[106] H. Zhao and P. Doshi. Towards automated restful web service composition. In *IEEE International Conference on Web Services (ICWS)*, 2009.

Appendix A

eDT-Golog

```
/*****************************************************************
%% eDT-Golog: an extension of DT-GOLOG

(DT-GOLOG: http://www.cs.toronto.edu/~cebly/Papers/dtgolog-abs.html)
*****************************************************************/


:- dynamic(proc/2).               /* Compiler directives. Be sure   */
:- set_flag(all_dynamic, on).     /* that you load this file first! */
:- set_flag(print_depth,500).
:- pragma(debug).

:- op(800, xfy, [&]).   /* Conjunction */
:- op(850, xfy, [v]).   /* Disjunction */
:- op(870, xfy, [=>]).  /* Implication */
:- op(880,xfy, [<=>]).  /* Equivalence */
:- op(950, xfy, [:]).   /* Action sequence */
:- op(960, xfy, [#]).   /* Nondeterministic action choice */


/*
To handle the representation of time-related reward and cost
We need to introduce integral predicates and the predicates
to compute the total rewards
*/


integralFunction(Y, X, Mean, Sigma, Alpha, do(A,S)) :-
    setAlpha(Alpha),
    meanValue(Mean, do(A,S)),
    sigmaValue(Sigma, do(A,S)),
    Y is (exp(-1*Alpha*X))*
    (1/(Sigma*sqrt(pi))*exp((-1)*(X-Mean)*(X-Mean)/(2*Sigma*Sigma))).
```

```
integral(Sum, do(A,S)) :- tmaxValue(Tmax, do(A,S)),
                  (for(X, 0, Tmax),fromto(0,In,Out,Sum)
                      do integralFunction(Y, X, Mean, Sigma, Alpha,
                                          do(A,S)), Out is In + Y ).


reward(TC, do(A,S)) :- rewardValue(R, do(A,S)),
                       lumpCostValue(K, do(A,S)),
                       costRate(C, do(A,S)),
                       integral(Sum, do(A,S)),
                       TC is R - K - C * Sum .



/* The predicate bp() is the top level call. Add an end-of-program
   marker "nil" to the tail of program expression E , then compute
   the best policy that succeeds with probability Prob. The horizon
   H must be a non-negative integer number.
*/


bp(E,H,Pol,Util,Prob,File) :- integer(H), H >= 0,
      cputime(StartT),
      bestDo(E : nil,s0,H,Pol,Val,Prob),
      cputime(EndT), Elapsed is EndT - StartT,
      Util is float(Val),
      open( File, append, Stream),
      date(Date),
      printf(Stream, "\n\nThis report is started at time %w\n", [Date]),
      (proc(E,Body) ->
       printf(Stream, "The Golog program is\n proc(%w,\n %w)\n",[E,Body]);
       printf(Stream, "The Golog program is\n %w\n",[E])
       ),
      printf("\nThe computation took %w seconds",[Elapsed]),
      printf(Stream, "\nTime elapsed is %w seconds\n", [Elapsed]),
      printf(Stream, "The optimal policy is \n %w \n", [Pol]),
      printf(Stream, "The value of the optimal policy is %w\n",[Util]),
      printf(Stream, "The probability of successful termination is %w\n",[Prob]),
      close(Stream).


bp2(E,H,Pol,Util,Prob,File, PolString) :- integer(H), H >= 0,
      cputime(StartT),
```

```
      bestDo(E : nil,s0,H,Pol,Val,Prob),
      cputime(EndT), Elapsed is EndT - StartT,
      Util is float(Val),

        term_to_string(Pol, PolString0),
        PolString = PolString0,

      open( File, append, Stream),
      date(Date),
      printf(Stream, "\n\nThis report is started at time %w\n", [Date]),
      ( proc(E,Body) ->
        printf(Stream, "The Golog program is\n proc(%w,\n %w)\n",[E,Body]);
        printf(Stream, "The Golog program is\n %w\n",[E])
       ),
      printf("\nThe computation took %w seconds",[Elapsed]),
      printf(Stream, "\nTime elapsed is %w seconds\n", [Elapsed]),
      printf(Stream, "The optimal policy is\n %w \n", [Pol]),
      printf(Stream, "The value of the optimal policy is %w\n",[Util]),
      printf(Stream, "The probability of successful termination is %w\n",[Prob]),

      close(Stream).



term_to_string(T, S) :-
    open(string(""), write, Stream),
    % use the flags which you want
    printf(Stream, "%w", [T]),
    get_stream_info(Stream, name, S),
    close(Stream).


/*  bestDo(E,S,H,Pol,V,Prob)
Given a Golog program E and situation S find a policy Pol of the
highest expected utility Val. The optimal policy covers a set of
alternative histories with the total probability Prob. H is a
given finite horizon.
*/


bestDo((E1 : E2) : E,S,H,Pol,V,Prob) :-   H >= 0,
           bestDo(E1 : (E2 : E),S,H,Pol,V,Prob).


bestDo(?(C) : E,S,H,Pol,V,Prob) :-   H >= 0,
```

```
          holds(C,S) ->  bestDo(E,S,H,Pol,V,Prob) ;
                 ( (Prob is 0.0) , Pol = stop, reward(V,S) ).


bestDo((E1 # E2) : E,S,H,Pol,V,Prob) :-  H >= 0,
     bestDo(E1 : E,S,H,Pol1,V1,Prob1),
     bestDo(E2 : E,S,H,Pol2,V2,Prob2),
     ( lesseq(V1,Prob1,V2,Prob2), Pol=Pol2, Prob=Prob2, V=V2 ;
       greatereq(V1,Prob1,V2,Prob2), Pol=Pol1, Prob=Prob1, V=V1).


bestDo(if(C,E1,E2) : E,S,H,Pol,V,Prob) :-   H >= 0,
               holds(C,S) -> bestDo(E1 : E,S,H,Pol,V,Prob) ;
                             bestDo(E2 : E,S,H,Pol,V,Prob).


bestDo(while(C,E1) : E,S,H,Pol,V,Prob) :-   H >= 0,
          holds(-C,S) -> bestDo(E,S,H,Pol,V,Prob) ;
                         bestDo(E1 : while(C,E1) : E,S,H,Pol,V,Prob).


bestDo(ProcName : E,S,H,Pol,V,Prob) :-   H >= 0,
              proc(ProcName,Body),
              bestDo(Body : E,S,H,Pol,V,Prob).

/* Non-decision theoretic version of pi: pick a fresh value of X
   and for this value do the complex action E1 followed by E.
*/
bestDo(pi(X,E1) : E,S,H,Pol,V,Prob) :-   H >= 0,
            sub(X,_,E1,E1_X), bestDo(E1_X : E,S,H,Pol,V,Prob).

/*
  Discrete version of pi. pickBest(x,f,e) means: choose the best value
   of x from the finite non-empty range of values f, and for this x,
   do the complex action expression e.
*/
bestDo(pickBest(X,F,E) : EF,S,H,Pol,V,Prob) :-  H >= 0,
        range(F,R),
        ( R=[D],    sub(X,D,E,E_D),
                bestDo(E_D : EF,S,H,Pol,V,Prob)           ;
          R=[D1,D2], sub(X,D1,E,E_D1), sub(X,D2,E,E_D2),
                bestDo((E_D1 # E_D2) : EF,S,H,Pol,V,Prob)  ;
          R=[D1,D2 | Tail], Tail = [D3 | Rest],
                sub(X,D1,E,E_D1), sub(X,D2,E,E_D2),
                bestDo((E_D1 # E_D2 # pickBest(X,Tail,E)) :
```

```
                                          EF,S,H,Pol,V,Prob)
        ).

bestDo(A : E,S,H,Pol,V,Prob) :-   H > 0,
      agentAction(A), deterministic(A,S),
      ( not poss(A,S), Pol = stop, (Prob is 0.0) , reward(V,S) ;
        poss(A,S), Hor is H - 1,
        bestDo(E,do(A,S),Hor,RestPol,VF,Prob),
        reward(R,S),
        V is R + VF,
        ( RestPol = nil,     Pol = A ;
          not RestPol=nil,   Pol = (A : RestPol)
         )
       ).

bestDo(A : E,S,H,Pol,V,Prob) :-   H > 0,
       agentAction(A), nondetActions(A,S,NatOutcomesList),
       Hor is H -1,
       bestDoAux(NatOutcomesList,E,S,Hor,RestPol,VF,Prob),
       reward(R,S),
       V is R + VF,
       Pol=(A : senseEffect(A) : (RestPol)).



bestDoAux([N1],E,S,H,Pol,V,Prob) :-   H >= 0, senseCondition(N1,Phi1),
     ( not poss(N1,S), ( Pol= (?(Phi1) : stop), (Prob is 0.0) , V is 0 ) ;
       poss(N1,S),
       prob(N1,Pr1,S),
       bestDo(E,do(N1,S),H,Pol1,V1,Prob1), !,
       Pol = ?(Seg_Start, C_Start, Phi1, C_End : Pol1, Seg_End),
       V is Pr1*V1,
       Prob is Pr1*Prob1 ).



bestDoAux([N1 | OtherOutcomes],E,S,H,Pol,V,Prob) :-  H >= 0,
     OtherOutcomes = [Head | Tail],  % there is at least one other outcome
     ( not poss(N1,S) -> bestDoAux(OtherOutcomes,E,S,H,Pol,V,Prob) ;
       poss(N1,S),
       bestDoAux(OtherOutcomes,E,S,H,PolT,VT,ProbT),
       senseCondition(N1,Phi1),
       prob(N1,Pr1,S),
       bestDo(E,do(N1,S),H,Pol1,V1,Prob1), !,
```

```
       Pol = if(Seg_Start, C_Start, Phi1,  C_End, % then
                            Pol1, % else
                                          (PolT), Seg_End),
       V is VT + Pr1*V1,
       Prob is ProbT + Pr1*Prob1 ).



bestDo(nil,S,H,Pol,V,Prob) :-
               Pol=nil, reward(V,S), (Prob is 1.0) .

bestDo(nil : E,S,H,Pol,V,Prob) :- H > 0, bestDo(E,S,H,Pol,V,Prob).

bestDo(stop : E,S,H,Pol,V,Prob) :-
               Pol=stop, reward(V,S), (Prob is 0.0) .

bestDo(E,S,H,Pol,V,Prob) :- H =:= 0,
        /*  E=(A : Tail), agentAction(A),  */
              Pol=nil, reward(V,S), (Prob is 1.0) .


  /* ---- Some useful predicates mentioned in the interpreter ----- */

lesseq(V1,Prob1,V2,Prob2) :-  Pr1 is float(Prob1), (Pr1 = 0.0) ,
         Pr2 is float(Prob2),
         ( (Pr2 \= 0.0) ;
           (Pr2 = 0.0) ,  V1 =< V2
          ).

lesseq(V1,Prob1,V2,Prob2) :-
            (Prob1 \= 0.0) , (Prob2 \= 0.0) , V1 =< V2.

greatereq(V1,Prob1,V2,Prob2) :-   (Prob1 \= 0.0) , (Prob2 = 0.0) .

greatereq(V1,Prob1,V2,Prob2) :-
            (Prob1 \= 0.0) , (Prob2 \= 0.0) , V2 =< V1.


deterministic(A,S) :- not nondetActions(A,S,OutcomesList).


range([D | Tail],[D | Tail]).     % Tail can be []

/* sub(Name,New,Term1,Term2): Term2 is Term1 with Name
   replaced by New. */
```

```
sub(X1,X2,T1,T2) :- var(T1), T2 = T1.
sub(X1,X2,T1,T2) :- not var(T1), T1 = X1, T2 = X2.
sub(X1,X2,T1,T2) :- not T1 = X1, T1 =..[F|L1], sub_list(X1,X2,L1,L2),
                    T2 =..[F|L2].
sub_list(X1,X2,[],[]).
sub_list(X1,X2,[T1|L1],[T2|L2]) :- sub(X1,X2,T1,T2),
                                   sub_list(X1,X2,L1,L2).

/* The holds predicate implements the revised Lloyd-Topor
   transformations on test conditions.  */

holds(P & Q,S) :- holds(P,S), holds(Q,S).
holds(P v Q,S) :- holds(P,S); holds(Q,S).
holds(P => Q,S) :- holds(-P v Q,S).
holds(P <=> Q,S) :- holds((P => Q) & (Q => P),S).
holds(-(-P),S) :- holds(P,S).
holds(-(P & Q),S) :- holds(-P v -Q,S).
holds(-(P v Q),S) :- holds(-P & -Q,S).
holds(-(P => Q),S) :- holds(-(-P v Q),S).
holds(-(P <=> Q),S) :- holds(-((P => Q) & (Q => P)),S).
holds(-all(V,P),S) :- holds(some(V,-P),S).
holds(-some(V,P),S) :- not holds(some(V,P),S).  /* Negation */
holds(-P,S) :- isAtom(P), not holds(P,S).     /* by failure */
holds(all(V,P),S) :- holds(-some(V,-P),S).
holds(some(V,P),S) :- sub(V,_,P,P1), holds(P1,S).

/* The following clause treats the holds predicate for all atoms,
   including Prolog system predicates. For this to work properly,
   the GOLOG programmer must provide, for all atoms taking a
   situation argument, a clause giving the result of restoring
   its suppressed situation argument, for example:
          restoreSitArg(ontable(X),S,ontable(X,S)).             */

holds(A,S) :- restoreSitArg(A,S,F), F ;
           not restoreSitArg(A,S,F), isAtom(A), A.

isAtom(A) :- not (A = -W ; A = (W1 & W2) ; A = (W1 => W2) ;
    A = (W1 <=> W2) ; A = (W1 v W2) ; A = some(X,W) ; A = all(X,W)).
```

Appendix B

Planning Domains for the Processes in the Experiments

B.1 Planning Domain for the Process with 3 suppliers

```
%% Author: Haibo Zhao
%% Date: 2/14/2008

proc(search, (receiveOrder(Order) : verifyOrder(Order) #
( checkInventory(Order) # checkPreSupplier(Order)
# checkSpotMarket(Order) )# selectShipper(Order)
# getGoods(Order) # shipGoods(Order)) : search ).
/* Stochastic actions have a finite number of outcomes:
we list all of them
*/


nondetActions(receiveOrder(Order),S,
[receiveOrderS(Order), receiveOrderF(Order)]).
nondetActions(verifyOrder(Order),S,
[verifyOrderS(Order),verifyOrderF(Order)]).
nondetActions(checkInventory(Order),S,
[checkInventoryS(Order),checkInventoryF(Order)]).
nondetActions(checkPreSupplier(Order),S,
[checkPreSupplierS(Order),checkPreSupplierF(Order)]).
nondetActions(checkSpotMarket(Order),S,
[checkSpotMarketS(Order),checkSpotMarketF(Order)]).
nondetActions(selectShipper(Order),S,
[selectShipperS(Order),selectShipperF(Order)]).
nondetActions(getGoods(Order),S,
[getGoodsS(Order),getGoodsF(Order)]).
nondetActions(shipGoods(Order),S,
[shipGoodsS(Order),shipGoodsF(Order)]).


/* Using predicate prob(Outcome,Probability,Situation)
we specify numerical values of probabilities for each outcome
```

```
*/

prob(receiveOrderS(Order), 0.99, S).
prob(receiveOrderF(Order), 0.01, S).
prob(verifyOrderS(Order), 0.99, S).
prob(verifyOrderF(Order), 0.01, S).
prob(checkInventoryS(Order), 0.80, S).
prob(checkInventoryF(Order), 0.20, S).
prob(checkPreSupplierS(Order), 0.90, S).
prob(checkPreSupplierF(Order), 0.10, S).
prob(checkSpotMarketS(Order), 0.99, S).
prob(checkSpotMarketF(Order), 0.01, S).

prob(selectShipperS(Order), 0.99, S). prob(selectShipperF(Order), 0.01, S).
prob(getGoodsS(Order), 0.99, S).  prob(getGoodsF(Order), 0.01, S).
prob(shipGoodsS(Order), 0.99, S). prob(shipGoodsF(Order), 0.01, S).

/* We formulate precondition axioms using the predicate
poss(Outcome, Situation) The right-hand side of precondition
axioms provides conditions under which Outcome is possible
in Situation
*/

poss(receiveOrderS(Order),S).
poss(receiveOrderF(Order),S).
poss(verifyOrderS(Order),S) :- isOrderReceived(Order,S),
not isOrderVerified(Order,S).
poss(verifyOrderF(Order),S) :- isOrderReceived(Order,S),
not isOrderVerified(Order,S).
poss(checkInventoryS(Order),S) :- isOrderVerified(Order,S),
not isOrderSatisfied(Order,S).
poss(checkInventoryF(Order),S) :- isOrderVerified(Order,S),
not isOrderSatisfied(Order,S).
poss(checkPreSupplierS(Order),S) :- isOrderVerified(Order,S),
not isOrderSatisfied(Order,S).
poss(checkPreSupplierF(Order),S) :- isOrderVerified(Order,S),
not isOrderSatisfied(Order,S).
poss(checkSpotMarketS(Order),S) :- isOrderVerified(Order,S),
not isOrderSatisfied(Order,S).
poss(checkSpotMarketF(Order),S)  :- isOrderVerified(Order,S),
not isOrderSatisfied(Order,S).
poss(selectShipperS(Order),S) :- isOrderSatisfied(Order,S),
not isShipperSelected(Order,S).
poss(selectShipperF(Order),S) :- isOrderSatisfied(Order,S),
not isShipperSelected(Order,S).
```

```
poss(getGoodsS(Order),S) :- isShipperSelected(Order,S),
not isGoodsReceived(Order,S).
poss(getGoodsF(Order),S) :- isShipperSelected(Order,S),
not isGoodsReceived(Order,S).
poss(shipGoodsS(Order),S) :- isGoodsReceived(Order,S),
not isShipped(Order,S).
poss(shipGoodsF(Order),S)  :- isGoodsReceived(Order,S),
not isShipped(Order,S).


/* Successor state axioms */
isOrderReceived(Order, do(A,S)) :- A=receiveOrderS(Order);
isOrderReceived(Order,S).
isOrderVerified(Order, do(A,S)) :- A=verifyOrderS(Order);
isOrderVerified(Order,S).
isOrderSatisfied(Order, do(A,S)):- A=checkInventoryS(Order) ;
A=checkPreSupplierS(Order); A=checkSpotMarketS(Order); isOrderSatisfied(Order,S).
isShipperSelected(Order,do(A,S)):- A=selectShipperS(Order) ;
isShipperSelected(Order,S).
isGoodsReceived(Order,do(A,S)):- A=getGoodsS(Order) ;
isGoodsReceived(Order,S).
isShipped(Order,do(A,S)):- A=shipGoods(Order) ; isShipped(Order,S).

/*Modified Reward functions in Logic*/
% Discount factor exp(-Alpha) = 0.9
setAlpha(Alpha) :- Alpha is 0.11 .
reward(0,s0).

lumpCostValue(K,do(A,S)) :- A = receiveOrderF(Order),
(not isOrderReceived(Order,S), K is 10;
isOrderReceived(Order,S), K is 2 ).
lumpCostValue(K,do(A,S)) :- A = verifyOrderF(Order),
(isOrderReceived(Order,S), not isOrderVerified(Order,S),K is 10; K is 2).
lumpCostValue(K,do(A,S)) :- A = checkInventoryF(Order), K is 12.
lumpCostValue(K,do(A,S)) :- A = checkPreSupplierF(Order), K is 6 .
lumpCostValue(K,do(A,S)) :- A = checkSpotMarketF(Order), K is 2.
lumpCostValue(K,do(A,S)) :- A = selectShipperF(Order),
(isOrderReceived(Order,S), isOrderVerified(Order,S),
isOrderSatisfied(Order,S), not isShipperSelected(Order,S),K is 10;K is 2).
lumpCostValue(K,do(A,S)) :- A = getGoodsF(Order),(isOrderReceived(Order,S),
isOrderVerified(Order,S), isOrderSatisfied(Order,S),
isShipperSelected(Order,S), not isGoodsReceived(Order,S),K is 10; K is 2 ).
lumpCostValue(K,do(A,S)) :- A = shipGoodsF(Order),(isOrderReceived(Order,S),
isOrderVerified(Order,S), isOrderSatisfied(Order,S),
isShipperSelected(Order,S), isGoodsReceived(Order,S), not isShipped(Order,S),
```

```
K is 10;  K is 2 ).


lumpCostValue(K,do(A,S)) :- A = receiveOrderS(Order),
(not isOrderReceived(Order,S), K is 10;
isOrderReceived(Order,S), K is 2 ).
lumpCostValue(K,do(A,S)) :- A = verifyOrderS(Order),
(isOrderReceived(Order,S), not isOrderVerified(Order,S),
K is 10; K is 2 ).
lumpCostValue(K,do(A,S)) :- A = checkInventoryS(Order), K is 12 .
lumpCostValue(K,do(A,S)) :- A = checkPreSupplierS(Order), K is 6 .
lumpCostValue(K,do(A,S)) :- A = checkSpotMarketS(Order), K is 2.
lumpCostValue(K,do(A,S)) :- A = selectShipperS(Order),
(isOrderReceived(Order,S), isOrderVerified(Order,S),
isOrderSatisfied(Order,S), not isShipperSelected(Order,S),
K is 10;  K is 2 ).
lumpCostValue(K,do(A,S)) :- A = getGoodsS(Order),(isOrderReceived(Order,S),
isOrderVerified(Order,S), isOrderSatisfied(Order,S),
sisShipperSelected(Order,S), not isGoodsReceived(Order,S) ,
K is 10; K is 2).
lumpCostValue(K,do(A,S)) :- A = shipGoodsS(Order),(isOrderReceived(Order,S),
isOrderVerified(Order,S), isOrderSatisfied(Order,S),
isShipperSelected(Order,S), isGoodsReceived(Order,S), not isShipped(Order,S),
K is 10;  K is 2 ).


% Specify the accumulating rate, which only depends on the action to be performed
costRate(C, do(A,S)) :- A = receiveOrderS(Order), C is 2 .
costRate(C, do(A,S)) :- A = verifyOrderS(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkInventoryS(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkPreSupplierS(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkSpotMarketS(Order), C is 2 .
costRate(C, do(A,S)) :- A = selectShipperS(Order), C is 2 .
costRate(C, do(A,S)) :- A = getGoodsS(Order), C is 2 .
costRate(C, do(A,S)) :- A = shipGoodsS(Order), C is 2 .
costRate(C, do(A,S)) :- A = receiveOrderF(Order), C is 2 .
costRate(C, do(A,S)) :- A = verifyOrderF(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkInventoryF(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkPreSupplierF(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkSpotMarketF(Order), C is 2 .
costRate(C, do(A,S)) :- A = selectShipperF(Order), C is 2 .
costRate(C, do(A,S)) :- A = getGoodsF(Order), C is 2 .
costRate(C, do(A,S)) :- A = shipGoodsF(Order), C is 2 .
```

```
% Specify the maximal sojourn time
tmaxValue(Tmax, do(A,S)) :- A = receiveOrderS(Order), Tmax is 7 .
tmaxValue(Tmax, do(A,S)) :- A = verifyOrderS(Order), Tmax is 7 .
tmaxValue(Tmax, do(A,S)) :- A = checkInventoryS(Order), Tmax is 7 .
tmaxValue(Tmax, do(A,S)) :- A = checkPreSupplierS(Order), Tmax is 7 .
tmaxValue(Tmax, do(A,S)) :- A = checkSpotMarketS(Order), Tmax is 7 .
tmaxValue(Tmax, do(A,S)) :- A = selectShipperS(Order), Tmax is 7 .
tmaxValue(Tmax, do(A,S)) :- A = getGoodsS(Order), Tmax is 7 .
tmaxValue(Tmax, do(A,S)) :- A = shipGoodsS(Order), Tmax is 7 .
tmaxValue(Tmax, do(A,S)) :- A = receiveOrderF(Order), Tmax is 7 .
tmaxValue(Tmax, do(A,S)) :- A = verifyOrderF(Order), Tmax is 7 .
tmaxValue(Tmax, do(A,S)) :- A = checkInventoryF(Order), Tmax is 7 .
tmaxValue(Tmax, do(A,S)) :- A = checkPreSupplierF(Order), Tmax is 7 .
tmaxValue(Tmax, do(A,S)) :- A = checkSpotMarketF(Order), Tmax is 7 .
tmaxValue(Tmax, do(A,S)) :- A = selectShipperF(Order), Tmax is 7 .
tmaxValue(Tmax, do(A,S)) :- A = getGoodsF(Order), Tmax is 7 .
tmaxValue(Tmax, do(A,S)) :- A = shipGoodsF(Order), Tmax is 7 .


% Specify mean of the sojourn time distribution
meanValue(Mean, do(A,S)) :- A = receiveOrderS(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = verifyOrderS(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkInventoryS(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkPreSupplierS(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkSpotMarketS(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = selectShipperS(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = getGoodsS(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = shipGoodsS(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = receiveOrderF(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = verifyOrderF(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkInventoryF(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkPreSupplierF(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkSpotMarketF(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = selectShipperF(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = getGoodsF(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = shipGoodsF(Order), Mean is 3 .


% Specify the standard deviation of the sojourn time distribution
sigmaValue(Sigma, do(A,S)) :- A = receiveOrderS(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = verifyOrderS(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkInventoryS(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkPreSupplierS(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkSpotMarketS(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = selectShipperS(Order), Sigma is 1.0 .
```

```
sigmaValue(Sigma, do(A,S)) :- A = getGoodsS(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = shipGoodsS(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = receiveOrderF(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = verifyOrderF(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkInventoryF(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkPreSupplierF(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkSpotMarketF(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = selectShipperF(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = getGoodsF(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = shipGoodsF(Order), Sigma is 1.0 .

rewardValue(R, do(A,S)):- isShipped(Order,S), R is 100 ; R is 0 .

%goal(Order, S) :-isOrderReceived(Order,S) , isOrderVerified(Order,S).

/* The predicate senseCondition(Outcome,Psi) describes what logical
formula Psi should be evaluated to determine Outcome uniquely
*/
senseCondition(receiveOrderS(Order),isOrderReceived(Order)).
senseCondition(receiveOrderF(Order),(-isOrderReceived(Order))).
senseCondition(verifyOrderS(Order),isOrderVerified(Order)).
senseCondition(verifyOrderF(Order),(-isOrderVerified(Order))).
senseCondition(checkInventoryS(Order), (isOrderSatisfied(Order))).
senseCondition(checkInventoryF(Order), (-isOrderSatisfied(Order))).
senseCondition(checkPreSupplierS(Order), (isOrderSatisfied(Order))).
senseCondition(checkPreSupplierF(Order), (-isOrderSatisfied(Order))).
senseCondition(checkSpotMarketS(Order), (isOrderSatisfied(Order))).
senseCondition(checkSpotMarketF(Order), (-isOrderSatisfied(Order))).
senseCondition(selectShipperS(Order), (isShipperSelected(Order))).
senseCondition(selectShipperF(Order), (-isShipperSelected(Order))).
senseCondition(getGoodsS(Order), (isGoodsReceived(Order))).
senseCondition(getGoodsF(Order), (-isGoodsReceived(Order))).
senseCondition(shipGoodsS(Order), (isShipped(Order))).
senseCondition(shipGoodsF(Order), (-isShipped(Order))).

/* Agent actions vs natures actions: the former are those which can be
executed by agents, the latter (outcomes) can be executed only by nature
*/
agentAction(receiveOrder(Order)).
agentAction(verifyOrder(Order)).
agentAction(checkInventory(Order)).
agentAction(checkPreSupplier(Order)).
agentAction(checkSpotMarket(Order)).
agentAction(selectShipper(Order)).
agentAction(getGoods(Order)).
```

```
agentAction(shipGoods(Order)).

restoreSitArg(isOrderReceived(Order),S,isOrderReceived(Order,S)).
restoreSitArg(isOrderVerified(Order),S,isOrderVerified(Order,S)).
restoreSitArg(isOrderSatified(Order),S,isOrderSatified(Order,S)).
restoreSitArg(isShipperSelected(Order),S,isShipperSelected(Order,S)).
restoreSitArg(isGoodsReceived(Order),S,isGoodsReceived(Order,S)).
restoreSitArg(isShipped(Order),S,isShipped(Order,S)).
```

B.2 Planning Domain for the Process with 15 suppliers

```
%% Author: Haibo Zhao
%% Date: 2/18/2008

proc(search, (receiveOrder(Order) : verifyOrder(Order)#(checkInventory(Order)
# checkPreSupplier(Order) # checkSpotMarket(Order) # checkSupplier01(Order)
# checkSupplier02(Order) # checkSupplier03(Order) # checkSupplier04(Order)
# checkSupplier05(Order) # checkSupplier06(Order) # checkSupplier07(Order)
# checkSupplier08(Order) # checkSupplier09(Order) # checkSupplier10(Order)
# checkSupplier11(Order) # checkSupplier12(Order))# selectShipper(Order)
# getGoods(Order) # shipGoods(Order)) : search ).
/* Stochastic actions have a finite number of outcomes:
we list all of them
*/

nondetActions(receiveOrder(Order),S,[receiveOrderS(Order), receiveOrderF(Order)]).
nondetActions(verifyOrder(Order),S,[verifyOrderS(Order),
                                                    verifyOrderF(Order)]).
nondetActions(checkInventory(Order),S,[checkInventoryS(Order),
                                                    checkInventoryF(Order)]).
nondetActions(checkPreSupplier(Order),S,[checkPreSupplierS(Order),
                                                    checkPreSupplierF(Order)]).
nondetActions(checkSpotMarket(Order),S,[checkSpotMarketS(Order),
                                                    checkSpotMarketF(Order)]).
nondetActions(selectShipper(Order),S,[selectShipperS(Order),
                                                    selectShipperF(Order)]).
nondetActions(getGoods(Order),S,[getGoodsS(Order),getGoodsF(Order)]).
nondetActions(shipGoods(Order),S,[shipGoodsS(Order),shipGoodsF(Order)]).


/* Using predicate prob(Outcome,Probability,Situation)
we specify numerical values of probabilities for each outcome
*/
```

```
prob(receiveOrderS(Order), 0.99, S). prob(receiveOrderF(Order), 0.01, S).
prob(verifyOrderS(Order), 0.99, S).  prob(verifyOrderF(Order), 0.01, S).

prob(checkInventoryS(Order), 0.80, S).  prob(checkInventoryF(Order), 0.20, S).
prob(checkPreSupplierS(Order), 0.90, S). prob(checkPreSupplierF(Order), 0.10, S).
prob(checkSpotMarketS(Order), 0.99, S). prob(checkSpotMarketF(Order), 0.01, S).

prob(selectShipperS(Order), 0.99, S).  prob(selectShipperF(Order), 0.01, S).
prob(getGoodsS(Order), 0.99, S).  prob(getGoodsF(Order), 0.01, S).
prob(shipGoodsS(Order), 0.99, S). prob(shipGoodsF(Order), 0.01, S).

/* We formulate precondition axioms using the predicate
poss(Outcome, Situation)
The right-hand side of precondition axioms provides conditions
under which Outcome is possible in Situation
*/

poss(receiveOrderS(Order),S):-  not isOrderReceived(Order,S).
poss(receiveOrderF(Order),S):-  not isOrderReceived(Order,S).
poss(verifyOrderS(Order),S) :- isOrderReceived(Order,S),
                               not isOrderVerified(Order,S).
poss(verifyOrderF(Order),S) :- isOrderReceived(Order,S),
                               not isOrderVerified(Order,S).
poss(checkInventoryS(Order),S) :- isOrderVerified(Order,S),
                               not isOrderSatisfied(Order,S).
poss(checkInventoryF(Order),S) :- isOrderVerified(Order,S),
                               not isOrderSatisfied(Order,S).
poss(checkPreSupplierS(Order),S) :- isOrderVerified(Order,S),
                               not isOrderSatisfied(Order,S).
poss(checkPreSupplierF(Order),S) :- isOrderVerified(Order,S),
                               not isOrderSatisfied(Order,S).
poss(checkSpotMarketS(Order),S) :- isOrderVerified(Order,S),
                               not isOrderSatisfied(Order,S).
poss(checkSpotMarketF(Order),S)  :- isOrderVerified(Order,S),
                               not isOrderSatisfied(Order,S).
poss(selectShipperS(Order),S) :- isOrderSatisfied(Order,S),
                               not isShipperSelected(Order,S).
poss(selectShipperF(Order),S) :- isOrderSatisfied(Order,S),
                               not isShipperSelected(Order,S).
poss(getGoodsS(Order),S) :- isShipperSelected(Order,S),
                               not isGoodsReceived(Order,S).
poss(getGoodsF(Order),S) :- isShipperSelected(Order,S),
                               not isGoodsReceived(Order,S).
poss(shipGoodsS(Order),S) :- isGoodsReceived(Order,S),
```

```
                                not isShipped(Order,S).
poss(shipGoodsF(Order),S)  :- isGoodsReceived(Order,S),
                                not isShipped(Order,S).



/* Successor state axioms */
isOrderReceived(Order, do(A,S)) :- A=receiveOrderS(Order); isOrderReceived(Order,S).
isOrderVerified(Order, do(A,S)) :- A=verifyOrderS(Order); isOrderVerified(Order,S).
isOrderSatisfied(Order, do(A,S)):- A=checkInventoryS(Order);
A=checkPreSupplierS(Order); A=checkSpotMarketS(Order);
A=checkSupplier01S(Order); A=checkSupplier02S(Order);
A=checkSupplier03S(Order); A=checkSupplier04S(Order);
A=checkSupplier05S(Order); A=checkSupplier06S(Order);
A=checkSupplier07S(Order); A=checkSupplier08S(Order);
A=checkSupplier09S(Order); A=checkSupplier10S(Order);
A=checkSupplier11S(Order); A=checkSupplier12S(Order);
isOrderSatisfied(Order,S).

isShipperSelected(Order,do(A,S)):- A=selectShipperS(Order);
isShipperSelected(Order,S).
isGoodsReceived(Order,do(A,S)):- A=getGoodsS(Order);
isGoodsReceived(Order,S).
isShipped(Order,do(A,S)):- A=shipGoods(Order);isShipped(Order,S).


/*Modified Reward functions in Logic*/


% Discount factor exp(-Alpha) = 0.9
setAlpha(Alpha) :- Alpha is 0.11 .

reward(0,s0).

lumpCostValue(K,do(A,S)) :- A = receiveOrderF(Order),
(not isOrderReceived(Order,S), K is 10;
isOrderReceived(Order,S), K is 2 ).
lumpCostValue(K,do(A,S)) :- A = verifyOrderF(Order),
(isOrderReceived(Order,S), not isOrderVerified(Order,S),
K is 10; K is 2).
lumpCostValue(K,do(A,S)) :- A = checkInventoryF(Order),(isOrderReceived(Order,S),
isOrderVerified(Order,S),
not isOrderSatisfied(Order,S), K is 12;  K is 12 ).
lumpCostValue(K,do(A,S)) :- A = checkPreSupplierF(Order),(isOrderReceived(Order,S),
isOrderVerified(Order,S), not isOrderSatisfied(Order,S),
```

```
K is 6;  K is 6 ).
lumpCostValue(K,do(A,S)) :- A = checkSpotMarketF(Order),(isOrderReceived(Order,S),
isOrderVerified(Order,S), not isOrderSatisfied(Order,S),
K is 2;  K is 2 ).
lumpCostValue(K,do(A,S)) :- A = selectShipperF(Order),(isOrderReceived(Order,S),
isOrderVerified(Order,S),
isOrderSatisfied(Order,S), not isShipperSelected(Order,S),
K is 10;  K is 2 ).
lumpCostValue(K,do(A,S)) :-
A = getGoodsF(Order),(isOrderReceived(Order,S),
isOrderVerified(Order,S), isOrderSatisfied(Order,S),
isShipperSelected(Order,S), not isGoodsReceived(Order,S),
K is 10; K is 2 ).
lumpCostValue(K,do(A,S)) :- A = shipGoodsF(Order),(isOrderReceived(Order,S),
isOrderVerified(Order,S), isOrderSatisfied(Order,S),
isShipperSelected(Order,S), isGoodsReceived(Order,S), not isShipped(Order,S),
K is 10;  K is 2 ).


lumpCostValue(K,do(A,S)) :- A = receiveOrderS(Order),
(not isOrderReceived(Order,S), K is 10; isOrderReceived(Order,S), K is 2 ).
lumpCostValue(K,do(A,S)) :- A = verifyOrderS(Order),(isOrderReceived(Order,S),
not isOrderVerified(Order,S), K is 10; K is 2 ).
lumpCostValue(K,do(A,S)) :- A = checkInventoryS(Order),
(isOrderReceived(Order,S), isOrderVerified(Order,S),
not isOrderSatisfied(Order,S), K is 12;  K is 12 ).
lumpCostValue(K,do(A,S)) :- A =checkPreSupplierS(Order),
(isOrderReceived(Order,S), isOrderVerified(Order,S),
not isOrderSatisfied(Order,S), K is 6;  K is 6 ).
lumpCostValue(K,do(A,S)) :- A =
checkSpotMarketS(Order),(isOrderReceived(Order,S), isOrderVerified(Order,S),
not isOrderSatisfied(Order,S), K is 2;  K is 2 ).
lumpCostValue(K,do(A,S)) :- A =
selectShipperS(Order),(isOrderReceived(Order,S), isOrderVerified(Order,S),
isOrderSatisfied(Order,S), not isShipperSelected(Order,S), K is 10;  K is 2 ).
lumpCostValue(K,do(A,S)) :- A = getGoodsS(Order),(isOrderReceived(Order,S),
isOrderVerified(Order,S), isOrderSatisfied(Order,S),
isShipperSelected(Order,S), not isGoodsReceived(Order,S) ,K is 10; K is 2).
lumpCostValue(K,do(A,S)) :- A = shipGoodsS(Order),(isOrderReceived(Order,S),
isOrderVerified(Order,S), isOrderSatisfied(Order,S),
isShipperSelected(Order,S), isGoodsReceived(Order,S), not isShipped(Order,S),
K is 10;  K is 2 ).


% Specify the accumulating rate, which only depends on the action to be performed
```

```
costRate(C, do(A,S)) :- A = receiveOrderS(Order), C is 2 .
costRate(C, do(A,S)) :- A = verifyOrderS(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkInventoryS(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkPreSupplierS(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkSpotMarketS(Order), C is 2 .
costRate(C, do(A,S)) :- A = selectShipperS(Order), C is 2 .
costRate(C, do(A,S)) :- A = getGoodsS(Order), C is 2 .
costRate(C, do(A,S)) :- A = shipGoodsS(Order), C is 2 .

costRate(C, do(A,S)) :- A = receiveOrderF(Order), C is 2 .
costRate(C, do(A,S)) :- A = verifyOrderF(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkInventoryF(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkPreSupplierF(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkSpotMarketF(Order), C is 2 .
costRate(C, do(A,S)) :- A = selectShipperF(Order), C is 2 .
costRate(C, do(A,S)) :- A = getGoodsF(Order), C is 2 .
costRate(C, do(A,S)) :- A = shipGoodsF(Order), C is 2 .

% Specify the maximal sojourn time
tmaxValue(Tmax, do(A,S)) :- A = receiveOrderS(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = verifyOrderS(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = checkInventoryS(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = checkPreSupplierS(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = checkSpotMarketS(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = selectShipperS(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = getGoodsS(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = shipGoodsS(Order), Tmax is 5 .

tmaxValue(Tmax, do(A,S)) :- A = receiveOrderF(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = verifyOrderF(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = checkInventoryF(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = checkPreSupplierF(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = checkSpotMarketF(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = selectShipperF(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = getGoodsF(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = shipGoodsF(Order), Tmax is 5 .


% Specify mean of the sojourn time distribution
meanValue(Mean, do(A,S)) :- A = receiveOrderS(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = verifyOrderS(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkInventoryS(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkPreSupplierS(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkSpotMarketS(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = selectShipperS(Order), Mean is 3 .
```

```
meanValue(Mean, do(A,S)) :- A = getGoodsS(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = shipGoodsS(Order), Mean is 3 .

meanValue(Mean, do(A,S)) :- A = receiveOrderF(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = verifyOrderF(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkInventoryF(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkPreSupplierF(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkSpotMarketF(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = selectShipperF(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = getGoodsF(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = shipGoodsF(Order), Mean is 3 .

% Specify the standard deviation of the sojourn time distribution
sigmaValue(Sigma, do(A,S)) :- A = receiveOrderS(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = verifyOrderS(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkInventoryS(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkPreSupplierS(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkSpotMarketS(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = selectShipperS(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = getGoodsS(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = shipGoodsS(Order), Sigma is 1.0 .

sigmaValue(Sigma, do(A,S)) :- A = receiveOrderF(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = verifyOrderF(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkInventoryF(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkPreSupplierF(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkSpotMarketF(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = selectShipperF(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = getGoodsF(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = shipGoodsF(Order), Sigma is 1.0 .


rewardValue(R, do(A,S)):- isShipped(Order,S), R is 100 ; R is 0 .

%goal(Order, S) :-isOrderReceived(Order,S) , isOrderVerified(Order,S).

/* The predicate senseCondition(Outcome,Psi) describes what logical
formula Psi should be evaluated to determine Outcome uniquely
*/
senseCondition(receiveOrderS(Order),isOrderReceived(Order)).
senseCondition(receiveOrderF(Order),(-isOrderReceived(Order))).

senseCondition(verifyOrderS(Order),isOrderVerified(Order)).
senseCondition(verifyOrderF(Order),(-isOrderVerified(Order))).
```

```
senseCondition(checkInventoryS(Order), (isOrderSatisfied(Order))).
senseCondition(checkInventoryF(Order), (-isOrderSatisfied(Order))).

senseCondition(checkPreSupplierS(Order), (isOrderSatisfied(Order))).
senseCondition(checkPreSupplierF(Order), (-isOrderSatisfied(Order))).

senseCondition(checkSpotMarketS(Order), (isOrderSatisfied(Order))).
senseCondition(checkSpotMarketF(Order), (-isOrderSatisfied(Order))).

senseCondition(selectShipperS(Order), (isShipperSelected(Order))).
senseCondition(selectShipperF(Order), (-isShipperSelected(Order))).

senseCondition(getGoodsS(Order), (isGoodsReceived(Order))).
senseCondition(getGoodsF(Order), (-isGoodsReceived(Order))).

senseCondition(shipGoodsS(Order), (isShipped(Order))).
senseCondition(shipGoodsF(Order), (-isShipped(Order))).


/* Agent actions vs natures actions: the former are those which can be
executed by agents, the latter (outcomes) can be executed only by nature
*/
agentAction(receiveOrder(Order)).
agentAction(verifyOrder(Order)).
agentAction(checkInventory(Order)).
agentAction(checkPreSupplier(Order)).
agentAction(checkSpotMarket(Order)).
agentAction(selectShipper(Order)).
agentAction(getGoods(Order)).
agentAction(shipGoods(Order)).

restoreSitArg(isOrderReceived(Order),S,isOrderReceived(Order,S)).
restoreSitArg(isOrderVerified(Order),S,isOrderVerified(Order,S)).
restoreSitArg(isOrderSatified(Order),S,isOrderSatified(Order,S)).
restoreSitArg(isShipperSelected(Order),S,isShipperSelected(Order,S)).
restoreSitArg(isGoodsReceived(Order),S,isGoodsReceived(Order,S)).
restoreSitArg(isShipped(Order),S,isShipped(Order,S)).


nondetActions(checkSupplier01(Order),S,
[checkSupplier01S(Order),checkSupplier01F(Order)]).
prob(checkSupplier01S(Order), 0.85, S).
prob(checkSupplier01F(Order), 0.15, S).
poss(checkSupplier01S(Order),S) :- isOrderVerified(Order,S), not
isOrderSatisfied(Order,S).
```

```
poss(checkSupplier01F(Order),S) :- isOrderVerified(Order,S), not
isOrderSatisfied(Order,S).
lumpCostValue(K,do(A,S)) :- A = checkSupplier01S(Order), K is 12.
lumpCostValue(K,do(A,S)) :- A = checkSupplier01F(Order), K is 12.
costRate(C, do(A,S)) :- A = checkSupplier01S(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkSupplier01F(Order), C is 2 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier01S(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier01F(Order), Tmax is 5 .
meanValue(Mean, do(A,S)) :- A = checkSupplier01S(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkSupplier01F(Order), Mean is 3 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier01S(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier01F(Order), Sigma is 1.0 .
senseCondition(checkSupplier01S(Order), (isOrderSatisfied(Order))).
senseCondition(checkSupplier01F(Order), (-isOrderSatisfied(Order))).
agentAction(checkSupplier01(Order)).


nondetActions(checkSupplier02(Order),S,
[checkSupplier02S(Order),checkSupplier02F(Order)]).
prob(checkSupplier02S(Order), 0.85, S).
prob(checkSupplier02F(Order), 0.15, S).
poss(checkSupplier02S(Order),S) :- isOrderVerified(Order,S), not
isOrderSatisfied(Order,S).
poss(checkSupplier02F(Order),S) :- isOrderVerified(Order,S), not
isOrderSatisfied(Order,S).
lumpCostValue(K,do(A,S)) :- A = checkSupplier02S(Order), K is 12 .
lumpCostValue(K,do(A,S)) :- A = checkSupplier02F(Order), K is 12 .
costRate(C, do(A,S)) :- A = checkSupplier02S(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkSupplier02F(Order), C is 2 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier02S(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier02F(Order), Tmax is 5 .
meanValue(Mean, do(A,S)) :- A = checkSupplier02S(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkSupplier02F(Order), Mean is 3 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier02S(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier02F(Order), Sigma is 1.0 .
senseCondition(checkSupplier02S(Order), (isOrderSatisfied(Order))).
senseCondition(checkSupplier02F(Order), (-isOrderSatisfied(Order))).
agentAction(checkSupplier02(Order)).

nondetActions(checkSupplier03(Order),S,
[checkSupplier03S(Order),checkSupplier03F(Order)]).
prob(checkSupplier03S(Order), 0.85, S).
prob(checkSupplier03F(Order), 0.15, S).
poss(checkSupplier03S(Order),S) :- isOrderVerified(Order,S), not
isOrderSatisfied(Order,S).
```

```
poss(checkSupplier03F(Order),S) :- isOrderVerified(Order,S), not
isOrderSatisfied(Order,S).
lumpCostValue(K,do(A,S)) :- A = checkSupplier03S(Order), K is 12 .
lumpCostValue(K,do(A,S)) :- A = checkSupplier03F(Order), K is 12 .
costRate(C, do(A,S)) :- A = checkSupplier03S(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkSupplier03F(Order), C is 2 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier03S(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier03F(Order), Tmax is 5 .
meanValue(Mean, do(A,S)) :- A = checkSupplier03S(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkSupplier03F(Order), Mean is 3 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier03S(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier03F(Order), Sigma is 1.0 .
senseCondition(checkSupplier03S(Order), (isOrderSatisfied(Order))).
senseCondition(checkSupplier03F(Order), (-isOrderSatisfied(Order))).
agentAction(checkSupplier03(Order)).

nondetActions(checkSupplier04(Order),S,
[checkSupplier04S(Order),checkSupplier04F(Order)]).
prob(checkSupplier04S(Order), 0.85, S).
prob(checkSupplier04F(Order), 0.15, S).
poss(checkSupplier04S(Order),S) :- isOrderVerified(Order,S), not
isOrderSatisfied(Order,S).
poss(checkSupplier04F(Order),S) :- isOrderVerified(Order,S), not
isOrderSatisfied(Order,S).
lumpCostValue(K,do(A,S)) :- A = checkSupplier04S(Order), K is 12 .
lumpCostValue(K,do(A,S)) :- A = checkSupplier04F(Order), K is 12 .
costRate(C, do(A,S)) :- A = checkSupplier04S(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkSupplier04F(Order), C is 2 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier04S(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier04F(Order), Tmax is 5 .
meanValue(Mean, do(A,S)) :- A = checkSupplier04S(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkSupplier04F(Order), Mean is 3 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier04S(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier04F(Order), Sigma is 1.0 .
senseCondition(checkSupplier04S(Order), (isOrderSatisfied(Order))).
senseCondition(checkSupplier04F(Order), (-isOrderSatisfied(Order))).
agentAction(checkSupplier04(Order)).


nondetActions(checkSupplier05(Order),S,
[checkSupplier05S(Order),checkSupplier05F(Order)]).
prob(checkSupplier05S(Order), 0.85, S).
prob(checkSupplier05F(Order), 0.15, S).
poss(checkSupplier05S(Order),S) :- isOrderVerified(Order,S), not
isOrderSatisfied(Order,S).
```

```
poss(checkSupplier05F(Order),S) :- isOrderVerified(Order,S), not
isOrderSatisfied(Order,S).
lumpCostValue(K,do(A,S)) :- A = checkSupplier05S(Order), K is 12 .
lumpCostValue(K,do(A,S)) :- A = checkSupplier05F(Order), K is 12 .
costRate(C, do(A,S)) :- A = checkSupplier05S(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkSupplier05F(Order), C is 2 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier05S(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier05F(Order), Tmax is 5 .
meanValue(Mean, do(A,S)) :- A = checkSupplier05S(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkSupplier05F(Order), Mean is 3 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier05S(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier05F(Order), Sigma is 1.0 .
senseCondition(checkSupplier05S(Order), (isOrderSatisfied(Order))).
senseCondition(checkSupplier05F(Order), (-isOrderSatisfied(Order))).
agentAction(checkSupplier05(Order)).


nondetActions(checkSupplier06(Order),S,
[checkSupplier06S(Order),checkSupplier06F(Order)]).
prob(checkSupplier06S(Order), 0.85, S).
prob(checkSupplier06F(Order), 0.15, S).
poss(checkSupplier06S(Order),S) :- isOrderVerified(Order,S), not
isOrderSatisfied(Order,S).
poss(checkSupplier06F(Order),S) :- isOrderVerified(Order,S), not
isOrderSatisfied(Order,S).
lumpCostValue(K,do(A,S)) :- A =checkSupplier06S(Order),
(isOrderReceived(Order,S), isOrderVerified(Order,S),
not isOrderSatisfied(Order,S), K is 12;  K is 12 ).
lumpCostValue(K,do(A,S)) :- A =
checkSupplier06F(Order),(isOrderReceived(Order,S), isOrderVerified(Order,S),
not isOrderSatisfied(Order,S), K is 12;  K is 12 ).
costRate(C, do(A,S)) :- A = checkSupplier06S(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkSupplier06F(Order), C is 2 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier06S(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier06F(Order), Tmax is 5 .
meanValue(Mean, do(A,S)) :- A = checkSupplier06S(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkSupplier06F(Order), Mean is 3 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier06S(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier06F(Order), Sigma is 1.0 .
senseCondition(checkSupplier06S(Order), (isOrderSatisfied(Order))).
senseCondition(checkSupplier06F(Order), (-isOrderSatisfied(Order))).
agentAction(checkSupplier06(Order)).


nondetActions(checkSupplier07(Order),S,
```

```
[checkSupplier07S(Order),checkSupplier07F(Order)]).
prob(checkSupplier07S(Order), 0.85, S).
prob(checkSupplier07F(Order), 0.15, S).
poss(checkSupplier07S(Order),S) :- isOrderVerified(Order,S), not
isOrderSatisfied(Order,S).
poss(checkSupplier07F(Order),S) :- isOrderVerified(Order,S), not
isOrderSatisfied(Order,S).
lumpCostValue(K,do(A,S)) :- A =
checkSupplier07S(Order),(isOrderReceived(Order,S), isOrderVerified(Order,S),
not isOrderSatisfied(Order,S), K is 12;  K is 12 ).
lumpCostValue(K,do(A,S)) :- A =
checkSupplier07F(Order),(isOrderReceived(Order,S), isOrderVerified(Order,S),
not isOrderSatisfied(Order,S), K is 12;  K is 12 ).
costRate(C, do(A,S)) :- A = checkSupplier07S(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkSupplier07F(Order), C is 2 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier07S(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier07F(Order), Tmax is 5 .
meanValue(Mean, do(A,S)) :- A = checkSupplier07S(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkSupplier07F(Order), Mean is 3 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier07S(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier07F(Order), Sigma is 1.0 .
senseCondition(checkSupplier07S(Order), (isOrderSatisfied(Order))).
senseCondition(checkSupplier07F(Order), (-isOrderSatisfied(Order))).
agentAction(checkSupplier07(Order)).

nondetActions(checkSupplier08(Order),S,
[checkSupplier08S(Order),checkSupplier08F(Order)]).
prob(checkSupplier08S(Order), 0.85, S).
prob(checkSupplier08F(Order), 0.15, S).
poss(checkSupplier08S(Order),S) :- isOrderVerified(Order,S), not
isOrderSatisfied(Order,S).
poss(checkSupplier08F(Order),S) :- isOrderVerified(Order,S), not
 isOrderSatisfied(Order,S).
lumpCostValue(K,do(A,S)) :- A =
checkSupplier08S(Order),(isOrderReceived(Order,S), isOrderVerified(Order,S),
not isOrderSatisfied(Order,S), K is 12;  K is 12 ).
lumpCostValue(K,do(A,S)) :- A =
checkSupplier08F(Order),(isOrderReceived(Order,S), isOrderVerified(Order,S),
not isOrderSatisfied(Order,S), K is 12;  K is 12 ).
costRate(C, do(A,S)) :- A = checkSupplier08S(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkSupplier08F(Order), C is 2 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier08S(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier08F(Order), Tmax is 5 .
meanValue(Mean, do(A,S)) :- A = checkSupplier08S(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkSupplier08F(Order), Mean is 3 .
```

```
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier08S(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier08F(Order), Sigma is 1.0 .
senseCondition(checkSupplier08S(Order), (isOrderSatisfied(Order))).
senseCondition(checkSupplier08F(Order), (-isOrderSatisfied(Order))).
agentAction(checkSupplier08(Order)).


nondetActions(checkSupplier09(Order),S,
[checkSupplier09S(Order),checkSupplier09F(Order)]).
prob(checkSupplier09S(Order), 0.85, S).
prob(checkSupplier09F(Order), 0.15, S).
poss(checkSupplier09S(Order),S) :-
isOrderVerified(Order,S), not isOrderSatisfied(Order,S).
poss(checkSupplier09F(Order),S) :-
isOrderVerified(Order,S), not isOrderSatisfied(Order,S).
lumpCostValue(K,do(A,S)) :-
A = checkSupplier09S(Order),(isOrderReceived(Order,S),
isOrderVerified(Order,S), not isOrderSatisfied(Order,S), K is 12;  K is 12 ).
lumpCostValue(K,do(A,S)) :- A =
checkSupplier09F(Order),(isOrderReceived(Order,S), isOrderVerified(Order,S),
not isOrderSatisfied(Order,S), K is 12;  K is 12 ).
costRate(C, do(A,S)) :- A = checkSupplier09S(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkSupplier09F(Order), C is 2 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier09S(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier09F(Order), Tmax is 5 .
meanValue(Mean, do(A,S)) :- A = checkSupplier09S(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkSupplier09F(Order), Mean is 3 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier09S(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier09F(Order), Sigma is 1.0 .
senseCondition(checkSupplier09S(Order), (isOrderSatisfied(Order))).
senseCondition(checkSupplier09F(Order), (-isOrderSatisfied(Order))).
agentAction(checkSupplier09(Order)).

nondetActions(checkSupplier10(Order),S,
[checkSupplier10S(Order),checkSupplier10F(Order)]).
prob(checkSupplier10S(Order), 0.85, S).
prob(checkSupplier10F(Order), 0.15, S).
poss(checkSupplier10S(Order),S) :- isOrderVerified(Order,S),
not isOrderSatisfied(Order,S).
poss(checkSupplier10F(Order),S) :- isOrderVerified(Order,S), not
isOrderSatisfied(Order,S).
lumpCostValue(K,do(A,S)) :-
A =checkSupplier10S(Order),(isOrderReceived(Order,S), isOrderVerified(Order,S),
not isOrderSatisfied(Order,S), K is 12;  K is 12 ).
lumpCostValue(K,do(A,S)) :-
```

```
A = checkSupplier10F(Order),(isOrderReceived(Order,S),
isOrderVerified(Order,S), not isOrderSatisfied(Order,S), K is 12;  K is 12 ).
costRate(C, do(A,S)) :- A = checkSupplier10S(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkSupplier10F(Order), C is 2 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier10S(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier10F(Order), Tmax is 5 .
meanValue(Mean, do(A,S)) :- A = checkSupplier10S(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkSupplier10F(Order), Mean is 3 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier10S(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier10F(Order), Sigma is 1.0 .
senseCondition(checkSupplier10S(Order), (isOrderSatisfied(Order))).
senseCondition(checkSupplier10F(Order), (-isOrderSatisfied(Order))).
agentAction(checkSupplier10(Order)).


nondetActions(checkSupplier11(Order),S,
[checkSupplier11S(Order),checkSupplier11F(Order)]).
prob(checkSupplier11S(Order), 0.85, S).
prob(checkSupplier11F(Order), 0.15, S).
poss(checkSupplier11S(Order),S) :- isOrderVerified(Order,S),
not isOrderSatisfied(Order,S).
poss(checkSupplier11F(Order),S) :- isOrderVerified(Order,S),
not isOrderSatisfied(Order,S).
lumpCostValue(K,do(A,S)) :- A =
checkSupplier11S(Order),(isOrderReceived(Order,S), isOrderVerified(Order,S),
not isOrderSatisfied(Order,S), K is 12;  K is 12 ).
lumpCostValue(K,do(A,S)) :- A =
checkSupplier11F(Order),(isOrderReceived(Order,S), isOrderVerified(Order,S),
not isOrderSatisfied(Order,S), K is 12;  K is 12 ).
costRate(C, do(A,S)) :- A = checkSupplier11S(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkSupplier11F(Order), C is 2 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier11S(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier11F(Order), Tmax is 5 .
meanValue(Mean, do(A,S)) :- A = checkSupplier11S(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkSupplier11F(Order), Mean is 3 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier11S(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier11F(Order), Sigma is 1.0 .
senseCondition(checkSupplier11S(Order), (isOrderSatisfied(Order))).
senseCondition(checkSupplier11F(Order), (-isOrderSatisfied(Order))).
agentAction(checkSupplier11(Order)).


nondetActions(checkSupplier12(Order),S,
[checkSupplier12S(Order),checkSupplier12F(Order)]).
```

```
prob(checkSupplier12S(Order), 0.85, S).
prob(checkSupplier12F(Order), 0.15, S).
poss(checkSupplier12S(Order),S) :- isOrderVerified(Order,S),
not isOrderSatisfied(Order,S).
poss(checkSupplier12F(Order),S) :- isOrderVerified(Order,S),
not isOrderSatisfied(Order,S).
lumpCostValue(K,do(A,S)) :-
A = checkSupplier12S(Order),(isOrderReceived(Order,S),
isOrderVerified(Order,S), not isOrderSatisfied(Order,S), K is 12;  K is 12 ).
lumpCostValue(K,do(A,S)) :-
A = checkSupplier12F(Order),(isOrderReceived(Order,S),
isOrderVerified(Order,S), not isOrderSatisfied(Order,S), K is 12;  K is 12 ).
costRate(C, do(A,S)) :- A = checkSupplier12S(Order), C is 2 .
costRate(C, do(A,S)) :- A = checkSupplier12F(Order), C is 2 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier12S(Order), Tmax is 5 .
tmaxValue(Tmax, do(A,S)) :- A = checkSupplier12F(Order), Tmax is 5 .
meanValue(Mean, do(A,S)) :- A = checkSupplier12S(Order), Mean is 3 .
meanValue(Mean, do(A,S)) :- A = checkSupplier12F(Order), Mean is 3 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier12S(Order), Sigma is 1.0 .
sigmaValue(Sigma, do(A,S)) :- A = checkSupplier12F(Order), Sigma is 1.0 .
senseCondition(checkSupplier12S(Order), (isOrderSatisfied(Order))).
senseCondition(checkSupplier12F(Order), (-isOrderSatisfied(Order))).
agentAction(checkSupplier12(Order)).
```

Appendix C

OWL-RESTWS Ontology

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
<!ENTITY owl-restws  "http://lsdis.cs.uga.edu/owl-restws-v0.2/owl-restws#" >
<!ENTITY owl  "http://www.w3.org/2002/07/owl#" >
<!ENTITY xsd  "http://www.w3.org/2001/XMLSchema#" >
<!ENTITY ruleml "http://www.w3.org/2003/11/ruleml#">
<!ENTITY swrl "http://www.w3.org/2003/11/swrl#">
]>

<rdf:RDF
  xmlns     = "http://lsdis.cs.uga.edu/owl-restws-v0.2/owl-restws#"
  xmlns:owl-restws = "http://lsdis.cs.uga.edu/owl-restws-v0.2/owl-restws#"
  xml:base  =  "http://lsdis.cs.uga.edu/owl-restws-v0.2/owl-restws#"
  xmlns:owl =  "http://www.w3.org/2002/07/owl#"
  xmlns:rdf =  "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs=  "http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd =  "http://www.w3.org/2001/XMLSchema#"
  xmlns:ruleml= "http://www.w3.org/2003/11/ruleml#"
  xmlns:swrl=  "http://www.w3.org/2003/11/swrl#"
>

  <owl:Ontology rdf:about="">
    <rdfs:comment>An OWL ontology for describing RESTful Web services</rdfs:comment>
    <rdfs:comment>A preliminary attempt to describe RESTful Web services using
    owl</rdfs:comment>
    <rdfs:label>Owl Ontology for RESTful Web service </rdfs:label>
  </owl:Ontology>

  <owl:Class rdf:ID="WSResource">
   <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#name" />
      </owl:Restriction>
     </rdfs:subClassOf>
```

```
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#description" />
      </owl:Restriction>
     </rdfs:subClassOf>
    <rdfs:label xml:lang="en">Web service resources</rdfs:label>
  </owl:Class>


  <owl:Class rdf:ID="Individual-WSResource">
   <rdfs:subClassOf rdf:resource="#WSResource" />
   <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#domain_resource" />
<owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
      </owl:Restriction>
     </rdfs:subClassOf>
    <rdfs:label xml:lang="en">Individual WS resources</rdfs:label>
  </owl:Class>


  <owl:Class rdf:ID="Set-WSResource">
   <rdfs:subClassOf rdf:resource="#WSResource" />
   <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#domain_resource" />
        <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">1
        </owl:minCardinality>
      </owl:Restriction>
     </rdfs:subClassOf>
    <rdfs:label xml:lang="en">A set of WS resources</rdfs:label>
  </owl:Class>


  <owl:DatatypeProperty rdf:ID="name">
    <rdfs:domain rdf:resource="#WSResource" />
    <rdfs:range  rdf:resource="&xsd;string"/>
  </owl:DatatypeProperty>


  <owl:DatatypeProperty rdf:ID="description">
    <rdfs:domain rdf:resource="#WSResource" />
    <rdfs:range  rdf:resource="&xsd;string"/>
  </owl:DatatypeProperty>
```

```
  <owl:ObjectProperty rdf:ID="domain_resource">
    <rdfs:domain rdf:resource="#Individual-WSResource" />
    <rdfs:range  rdf:resource="rdfs:Resource"/>
  </owl:ObjectProperty >

  <owl:Class rdf:ID="WebService" />

  <owl:Class rdf:ID="Restws">
    <rdfs:subClassOf rdf:resource="#WebService" />
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#hasName" />
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#hasDescription" />
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#hasURI" />
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#get" />
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#put" />
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#delete" />
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#post" />
      </owl:Restriction>
</rdfs:subClassOf>
```

```
  <rdfs:label xml:lang="en">restws</rdfs:label>
</owl:Class>


<owl:Class rdf:ID="Restws-I">
  <rdfs:subClassOf rdf:resource="#Restws" />
   <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#associated_set_WSresource" />
      <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:label xml:lang="en">restws I</rdfs:label>
</owl:Class>


<owl:Class rdf:ID="Restws-II">
  <rdfs:subClassOf rdf:resource="#Restws" />
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#post" />
      <owl:hasValue rdf:datatype="&xsd;boolean">false</owl:hasValue>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#associated_individual_WSresource" />
      <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1
      </owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:label xml:lang="en">restws II</rdfs:label>
</owl:Class>


<owl:Class rdf:ID="Restws-III">
  <rdfs:subClassOf rdf:resource="#Restws" />
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#get" />
<owl:hasValue rdf:datatype="&xsd;boolean">false</owl:hasValue>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
```

```
      <owl:onProperty rdf:resource="#put" />
<owl:hasValue rdf:datatype="&xsd;boolean">false</owl:hasValue>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#delete" />
      <owl:hasValue rdf:datatype="&xsd;boolean">false</owl:hasValue>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#post" />
      <owl:hasValue rdf:datatype="&xsd;boolean">true</owl:hasValue>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#onPost" />
<owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:label xml:lang="en">restws III</rdfs:label>
</owl:Class>


<owl:DatatypeProperty rdf:ID="hasName">
  <rdfs:domain rdf:resource="#Restws" />
  <rdfs:range  rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>


<owl:DatatypeProperty rdf:ID="hasDescription">
  <rdfs:domain rdf:resource="#Restws" />
  <rdfs:range  rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>


<owl:DatatypeProperty rdf:ID="hasURI">
  <rdfs:domain rdf:resource="#Restws" />
  <rdfs:range  rdf:resource="&xsd;anyURI"/>
</owl:DatatypeProperty>


<owl:DatatypeProperty rdf:ID="get">
```

```
    <rdfs:domain rdf:resource="#Restws" />
    <rdfs:range  rdf:resource="&xsd;boolean"/>
  </owl:DatatypeProperty>


  <owl:DatatypeProperty rdf:ID="put">
    <rdfs:domain rdf:resource="#Restws" />
    <rdfs:range  rdf:resource="&xsd;boolean"/>
  </owl:DatatypeProperty>


  <owl:DatatypeProperty rdf:ID="get">
    <rdfs:domain rdf:resource="#Restws" />
    <rdfs:range  rdf:resource="&xsd;boolean"/>
  </owl:DatatypeProperty>


  <owl:DatatypeProperty rdf:ID="post">
    <rdfs:domain rdf:resource="#Restws" />
    <rdfs:range  rdf:resource="&xsd;boolean"/>
  </owl:DatatypeProperty>


  <owl:ObjectProperty rdf:ID="associated_set_WSresource">
    <rdfs:domain rdf:resource="#Restws-I" />
    <rdfs:range  rdf:resource="#Set-WSResource"/>
  </owl:ObjectProperty>


  <owl:ObjectProperty rdf:ID="associated_individual_WSresource">
    <rdfs:domain rdf:resource="#Restws-II" />
    <rdfs:range  rdf:resource="#Individual-WSResource"/>
  </owl:ObjectProperty>


  <owl:ObjectProperty rdf:ID="onPost">
    <rdf:type rdf:resource="&owl;FunctionalProperty" />
    <rdfs:domain rdf:resource="#Restws-III" />
    <rdfs:range  rdf:resource="&ruleml;imp" />
  </owl:ObjectProperty>

</rdf:RDF>
```