Using assumptions in service composition context

Zheng Lu
University of Wollongong

Using Assumptions in Service Composition Context

A thesis submitted in fulfillment of the
requirements for the award of the degree

Master of Computer Science by Research

from

UNIVERSITY OF WOLLONGONG

by

Zheng Lu

School of IT & Computer Science
June 2006

Dedicated to

My Parents

Declaration

This is to certify that the work reported in this thesis was done by the author, unless specified otherwise, and that no part of it has been submitted in a thesis to any other university or similar institution.

Zheng Lu
November 8, 2006

Abstract

Service composition aims to provide an efficient and accurate model of a service, based on which the global service oriented architecture (SOA) can be realized, allowing value added services to be generated on the fly. Unlike a traditional software module, which runs within a predictable domain, Web Services are autonomous software agents running in a heterogeneous execution environment. Because of distributed responsibilities, ownership and control, it is often not feasible to acquire all information needed for the service composition. These characteristics of autonomy and heterogeneity are fundamental to service oriented computing but make it inherently difficult to avoid service conflicts. To reason about and adapt to a changing environment, in this work, we will extend current OWL-S by introducing the concept of service assumptions which allow reasoning with incomplete information. Furthermore, together with the proposed service assumptions, a sequence of rule conditions are proposed to describe all permitted behaviors in service composition context.

Acknowledgments

I would like to express my gratitude to my supervisors and for their many insightful comments and thoughts that guided me to finish this research. I am also thankful to my other colleagues in Decision Systems Laboratory (DSL) for their valuable comments, supports, helps and encouragement during the process of completing this thesis as well as during the period of my master study.

List of Publications

This is a list of referred papers that is related to this research work.

- Zheng Lu, Shiyan Li and Aditya K. Ghose, Web Service Conflict Management, Proceedings of the First International Workshop on Design of Service-Oriented Applications (WDSOA'05), in conjunction with Third International Conference on Service Oriented Computing 2005, Amsterdam, The Netherlands, 2005.
- Zheng Lu, Aditya K. Ghose, Peter Hyland and Ying Guan, Using Assumptions in Service Composition, In Proceedings the 2006 IEEE International Conference on Services Computing, SCC 2006, Chicago, USA, September 2006. To Appear in proceedings of SCC 2006
- Zheng Lu, Aditya K. Ghose, Peter Hyland, Adopting Default Reasoning in Service Composition Context. To appear in Proceedings of The 4th IEEE European Conference on Web Services (ECOWS) (ECOWS 2006), Zurich, Switzerland, 2006. IEEE Computer Society Press.
- Zheng Lu, Shiyan Li and Aditya K. Ghose and Peter Hyland, Extending Semantic Web Service Description by Service Assumption, Web Intelligence Conference, Hong Kong 2006. To appear in Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence (WI'06). Hong Kong, China, 2006.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Web Services are self-contained, self-describing applications which not only perform the business functions on their own, but also are capable of engaging other web services in order to complete more complex business transactions. Service-Oriented Computing (SOC [54]) is the computing paradigm that utilizes services as fundamental elements for developing application solutions. To build the service model, SOC relies on the Service Oriented Architecture (SOA [55]) which is an emerging approach and addresses the requirements of loosely coupled, standards-based, and protocol-independent distributed computing. The nature of loosely-coupled system serves the needs of a business to adapt rapidly to changes in policies, business environment, product offerings, partnerships and regulatory requirements. Briefly speaking, SOA is a way of reorganizing software applications and infrastructure into a set of interacting services. The driving goal of SOA is to eliminate communication barriers so that applications can be integrated between enterprises and within enterprises, i.e. Web Service composition.

Web Service composition is the ability of one business to provide value-added services to its customers through the composition of basic Web Services, possibly offered by different companies [57]. By offering an integrated service i.e. a composite service, enterprises are enabled to continuously discover new opportunities to form alliances with other enterprises to share their cost, skills and resources. Despite its enormous potential to achieve the integration of heterogeneous systems, the dynamic Web Service Composition is still the object of ongoing research activity. Due to the problems

of distributed responsibilities, accountability, authority, ownership, and control etc, typically, it is unrealistic to acquire complete information from all parties involved in dynamic service composition. Making decisions about the role of various Wed Services based upon partial or incomplete information often fails to achieve consistency. In other words, the dynamic composition of Web Services entails consistency problems or conflicts.

1.1 Motivation

Conflict has been defined in [61] as "the interaction of interdependent people who perceive opposition of goals, aims, and values, and who see the other party as potentially interfering with the realization of these goals...". This highlights some general characteristics of conflict: interaction, interdependence, and incompatible goals. In addition, it has been demonstrated that conflict is common in group interactions [26, 27, 64]. Often, Web Service composition involves multiple independent parities, and during the process of Web Service composition, the interactions between these independent parities have to be carried out to locate and invoke services. Thus we can assume that any service composition involving more than one independent service providers may be subject to typical group conflicts [26, 64].

The following example of a travel agency is used to explain the service conflicts which may be caused by incompleteness of information during the dynamic service composition. Our example uses the often presented travel agency service package. A typical use case could involve arranging a trip consisting a hotel booking, a car rental and a sightseeing service. To simplify this use case, we assume this composite service is executed in a sequential manner (i.e. hotel booking service, then car rental service, finally sightseeing service). Assume that, when requesting this composite travel agency service, the user specifies his preferred car model, for example, a city car. Obviously, this car will be used for sightseeing, which is also generated as part of this composite

service. If the functionality matches the user's requirement, then the car rental service is invoked. In the real world, it is most likely that the car rental service providers have some service policy about usage of rental cars. However, when the car rental service is invoked, we don't have any information about what kinds of sightseeing plan might have been generated from the execution of the service, in other words, we don't know how the rented car will be used. The point here is that different sightseeing plans may be associated with different roads, and it may not be allowable for a rented car to drive on certain roads. For example, a desert dune exploration plan is dynamically generated from the sightseeing service and a city car is used for the desert dune exploration. Clearly, this is not an acceptable situation for either the car rental company or the customer.

The platform neutral nature of Web Services creates the opportunity for building composite services by dynamically composing the functionalities of existing atomic or complex services. What makes achieving the consistent service composition complicated is the fact that services can interact in complex ways. Different enterprises have distinct business objectives, rules and assumptions about using or providing a Web Service, especially in some rule or policy intensive enterprises. It is unrealistic to acquire complete information from all parties involved during dynamic service composition. In this proposed work, we will extend the current Semantic Web Service Description [4] by introducing the concept of "Service Assumption". Together with this proposed extension, we will define a formal mechanism for reasoning about incomplete knowledge during dynamic service composition and to address the service conflict issues.

The term conflict is used to cover any interference in one service's activities, needs or goals, caused by the activities of another service [26]. Handling conflict is one of the factors that determines whether a group of services can be composed together successfully. In our framework, detecting the inconsistency among the various services is the way to avoid the service conflict. Inconsistency here can be viewed as a state in which two or more overlapping elements of different services make assertions about

certain aspects of the service composition they describe which are not jointly satisfiable. We believe that the question of conflict between various services is highly relevant to any dynamic service composition. To assume absence of conflicts in service composition is naive. If service descriptions ignore issues of conflict in a dynamic service execution environment, then this ignorance about conflict becomes embedded in the underlying service composition. This ignorance may influence the style of service composition in unplanned ways, for instance, by restricting the means that the composition planner has of dealing with conflicts.

1.2 Contributions

Ontology Web Language for Services (OWL-S [4]), formerly known as DAML-S [3], is an upper ontology for services, aimed at achieving the automation of service discovery, invocation, composition and interoperation. OWL-S leverages the rich expressive power of OWL [23], together with its well-defined semantics, to provide richer descriptions of Web Services. Recently, Semantic Web Rule Language (SWRL) [38] has been proposed to define service process preconditions and effects, process control conditions and their contingent relationships in OWL-S. Though OWL-S is endowed with more expressive power and reasoning options when combined with SWRL, the description provided by a combination of OWLS and SWRL about service composition is still only a partial picture of the real world. Most of what we know about the world, when formalized, will yield an incomplete theory precisely because we cannot know everything - there are gaps in our knowledge [62]. In the same way, the ontology of services, is finite and incomplete. Thus, a service composition specified by OWL-S has to deal with partial or incomplete knowledge. The contributes I have made to the field of service composition, are based on the current Semantic Web Service Description as follows:

- Currently, OWL-S has no mechanism for handling incomplete information in service composition context and no method for reasoning about its side-effects.

We extended current OWL-S by introducing the concept of service assumptions which allow reasoning with incomplete information. The proposed extension is an attempt to bridging the gap between the semantic service description and the multiple operational domains involved in dynamic service composition.

- We explain the semantics of service assumptions and how service assumptions can facilitate the further automation of service composition planning.

- To allow service assumptions to be used in a more flexible way, we further classify service assumptions as Hard Assumptions and Soft Assumptions. To avoid the self-defeating problem, we also classify service assumptions as Transient Assumptions and Persistent Assumptions.

- Together with the proposed service assumptions, we offer a sequence of rules to be used in the process of service composition planning, which describes conditions required to achieve consistent service composition.

1.3 Organization of the Thesis

The next chapter of this thesis surveys the works from a number of related fields. Web Service is an emerging research area, and the importance of Web Services has been recognized and widely accepted by industry and in academic research. Our review about Web Services will be conducted from both industry and academic research perspectives. Additionally, an implementation of a Web Service is a software module, thus Web Services inherit principles and technology from software development[78]. To understand the software consistency management, our survey also includes the brief review about the inconsistency problem in software engineering literature.

Dynamic composition of services is a hard problem and it is not entirely clear which techniques can solve the problem best. One family of techniques that has been proposed for this task is service composition as planning. We will give an overview

of recent research efforts concerning automatic Web Service composition from the AI planning research community. Then we explain some basic concepts of default logics which support reasoning with incomplete information. In addition to introducing the reader to these diverse areas, the chapter discusses their relevance to the topic of the thesis.

In Chapter 3, we extend the current version of OWL-S by adopting service assumptions, explain the semantics of the service assumptions and classifications of service assumptions. In Chapter 4, firstly, we define the basic semantics for the planning-based service composition domain. Then, based on our proposed extensions to the current Semantic Web Service Description, a formal framework for reasoning about incomplete knowledge during the service composition planning process is presented. In Chapter 5, we look at the often presented example, travel agency package, as a service composition. Two scenarios are presented in which the service assumption as an extension to current Semantic Web Service Description can prove to be useful. Chapter 6 presents our conclusions, a discussion about the limitations of the proposed approach is given and an outline of how this work can progress further.

Chapter 2

Background

Despite their enormous potential to achieve integration of heterogeneous systems, the composition of services in dynamic environments is still the object of ongoing research activity, particularly, the consistency problems associated with the integration of Web Services. In this chapter, a brief literature review of the major areas related to our study will be presented. These areas include: basic Web Service concepts, Semantic Web Services, consistency management in software engineering, default logic and AI planning.

2.1 Web Service

Web Services are techniques for direct communications between one program and another across the Web. The basic idea of the service-oriented computing paradigm has evolved from various distributed computing approaches such as electronic data interchange, common object and request broker, etc. However, service-oriented computing [54, 78] is different from a traditional distributed computing approach in that it enables the software to be created on the fly through the use of loosely coupled, reusable software components.

Generally speaking, Web Services are a new breed of Web application. They are self-contained, self-describing, modular applications that can be published, located, and invoked across the web [41]. Based on the industry-standard Web Services Model

[41, 9], there are three basic roles involved in the Web Service Architecture:

- **Service Provider**: a Web Service provides some functionality on behalf of its owner. From an architectural perspective, a Service Provider is the network application platform that uses Internet protocols to advertise and provide services to Web Service Requesters.

- **Service Requester**: an entity which wishes to make use of existing Web Services. Usually, it initializes the message exchanges with the service providers. From an architecture perspective, the service requester is an application that uses Internet protocols to access the information and functionality made available by Web Service Providers.

- **Service Registry**: a Service registry is a centrally controlled searchable registry of service descriptions. Service Requesters find services and obtain binding information from the service descriptions held in Service Registry. The service binding can be either static binding, which is defined during service development, or dynamic binding, which occurs during service execution.

It is well known that the current dominant programming paradigm is the object-oriented programming which uses direct invocation method, i.e. the method must be provided by an object running at invocation time. On the other hand, service oriented computing adopts the find-bind-use as its operation model. To use an existing service, a service requester first searches a service registry for descriptions of all available services, then the target service is selected based on service functional and non-functional descriptions. Finally, the service description is used to bind with the service provider and invoke with the Web Service implementation. The find-bind-use model [10] allows for greater flexibility, especially in distributed environments. Based on the Web Services Model [41, 9], the Web Service life cycle includes following basic operations:

- **Publish**: to be accessible, the service description needs to be published so that the service requester can find it.

- **Find**: the service requester retrieves a service description directly or queries the service registry for the type of service required.
- **Bind**: eventually, a service needs to be invoked. In the bind operation, the binding details specified in the service description are used by a service requester to initiate an interaction with the service at runtime.

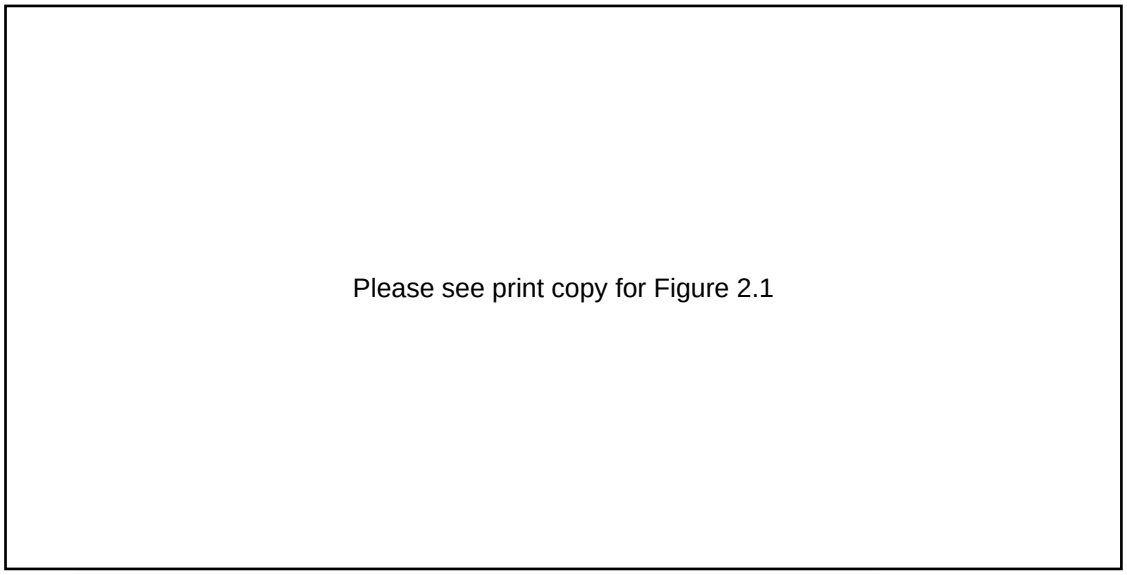

Figure 2.1: IBM Web Services Conceptual Stack

The Web Service architecture defined by IBM [41] is the most elaborate and the best described industry ad-hoc standard for Web Services, in which a Web Service is viewed as an abstract interface that specifies a set of network accessible operations through standardized XML messaging (See Fig 2.1). Layered on top of Network and XML-based messaging (SOAP), the service description layer is responsible for describing the interface of a Web Service and related service interactions. The current service description language supported by the industry is WSDL [6, 20], which is used to define the interface and mechanics of service interaction. SOAP offers a basic message exchange mechanism, while WSDL carries the information which indicates what messages should be exchanged during service execution. To enable service publishing and discovery, UDDI provides a mechanism for holding the descriptions of Web Services.

UDDI: The Universal Description, Discovery, and Integration specification defines a standard data structure for service descriptions, and specifies how to get access to the service publication and description. Thus, UDDI is responsible for both service publication and service discovery. Here, UDDI is roughly equivalent to an automated directory mechanism, but it also defines a data structure standard for representing service descriptions in a unified and systematic way.

One of most important characteristics of Web Services is the ability to be composable. Service composition is the top layer of the conceptual Web Service stack, aimed to support business-to-business or enterprise application integration. The Business Process Execution Language for Web Services (BPEL4WS[21]) is a language which aims to specify the common concepts for a business process execution language and help organizations coordinate business processes and transactions. The publication of BPEL4WS is a joint effort by BEA, IBM, Microsoft, SAP and Siebel, which superseded XLANG and WSFL as a standard for Web Services flow specification. BPEL4WS forms the necessary technical foundations for multiple usage patterns including both the process interface descriptions required for business protocols and executable process models. In addition, the underlying models provided by BPEL4WS collectively describe how to define, create, and connect multiple business processes in a Web Services environment. The BPEL4WS process model is layered on top of the service model defined by WSDL. The principle objective of BPEL4WS is to model the peer-to-peer interaction between services. The processes and the partners involved in a service composition are modeled by BPEL4WS as the standard service interface. Typically, these standard service interfaces are specified by using WSDL.

Basically, BPEL4WS supports two different ways of describing composite Web Service flow:

- Executable business processes: similar to workflow descriptions, the executable business processes are represented using basic and structured activities. They can be used to specify the details of business processes in a business interaction

as well as a pattern of execution of Web Services. Because of the explicitly specified internal details of process flows, the executable business processes can be executed by a service composition engine.

- Abstract business protocols: which only allow the specification of public message exchange between parties, and ignore process-internal data and computation.

BPEL4WS is essentially used to define a new Web Service by composing multiple existing services. Relying heavily on WSDL description of the Web Services, the interface of the new composite Web Service specified by BPEL4WS uses a set of port types specified by WSDL to provide combined operations like any other Web Service. Furthermore, the BPEL4WS process supports either sequential or parallel execution. The execution could be controlled by conditional expressions. Like the programming language, a BPEL4WS process has the ability to construct loops, declare variables, copy and assign values and define fault handlers. By combining all these constructs, the complex business processes could be defined in a fully controlled manner. A BPEL4WS process consists of a set of activities. Multiple primitive activities can be combined as a structured activity. Primitive activities represent basic constructs and are used for common tasks, such as the following:

- Invoke: used to invoke other Web Services.
- Receive: used to wait for the next invocation request of the business process.
- Reply: used to generate a response for synchronous operations.
- Assign: used to manipulate data variables.
- Throw: used to deal with faults and exceptions during service execution.
- Wait: used to wait for some time.
- Terminate: used to terminate the entire process.

In summary, from the industry respect, the base level of Web Service is WSDL which provides a standard Web Interface specification and provides a way to map from the abstract descriptions of a Web Service to its specific implementation. To describe other high level aspects of the Web Service, some other service descriptions are adopted to complement current WSDL. Together with WSDL, UDDI data structures are used to describe service context information, for example name and contact information of a service provider etc. At the top level of the Web Service conceptual stack, service compositions allow applications to be assembled from a set of appropriate existing software modules. BPEL4WS provides a straightforward way to compose several Web Services into a new composite service.

2.2 Semantic Web Service

Like we stated in the previous chapter, industry has made significant progress towards a new web application paradigm - service oriented computing. To provide a robust service development and operational environment, the current industry quasi-standard, however, is too flat to be comprehensive [69] and richer descriptions of Web Services are required. OWL Web Ontology Language for Services (OWL-S) Specification, formerly called DAML-S, was first developed by the DAML coalition [3] in 2001. The latest release is OWL-S version 1.1 (complete specification can be found at [4]). OWL-S is an upper ontology for services, aimed at achieving the automation of service discovery, invocation, composition and interoperation. OWL-S leverages the rich expressive power of OWL [23] together with its well-defined semantics to provide richer descriptions of Web Services. Service ontologies can be used to map service functional descriptions and domain properties into a standardized logic so that they can be machine understandable and interpretable. Thus OWL-S forms a good foundation for describing Web Services and their composition in an open, web-based environment.

The Semantic Web Service is an emerging research area which builds on the foundations of diverse prior works. First, the Semantic Web Service is built on the top of a new form of Web - the Semantic Web [16]. Second, the field of knowledge representation provides high-level descriptions of the world and logic foundation for Semantic Web Service Description, which can be used to characterize semantics and semantic relationships between various concepts. Finally, the service description languages proposed by industry such as WSDL defines a stateless client-server model of synchronous or uncorrelated asynchronous interactions. On the other hand, OWL-S, siting on top of these low-level communication layers, provides a richer semantic description of Web Services and tries to answer the different questions.

The Semantic Web is not a separate Web, but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation [16]. OWL-S adopts the Semantic Web technology for service description. Description Logics [14] and Ontology Web Language [23] provide the semantic foundation for the markup of Web Services. Description Logic [14] is sometimes called terminological logic, classification logic or concept logic. It is a technical thread of formal knowledge representations, aimed at providing a logic language for expressing and using semantic information. Generally speaking, Description Logic is a knowledge representation formalism, which provides structures for organizing and reasoning about the expression of knowledge and was developed to:

1. emphasize representing knowledge language.

2. define the common terminology in the application domain.

3. support the design of knowledge-based systems.

In order for information to be exchanged between service applications and for knowledge to reused in an open-ended environment, a shared understanding of the relevant domains are required. A solution to this problem is provided by ontologies, which aim to capture the semantics of a particular subject area. Basically, an ontology defines the

basic terms and relations comprising the vocabulary of a topic area, as well as the rules for combining terms and relations to define extensions to the vocabulary [47]. In the context of the Semantic Web, ontologies can encode the descriptions of web resources or services in a way that enables machines to use or process these descriptions. Usually, an ontology is represented in a logical language, for example, Description Logic. When an ontology is codified in a formal knowledge representation language, it is endowed with more expressive power and reasoning options, thus the description of an ontology is more precise and machine interpretable.

An ontology can be viewed as a finite controlled but extensible vocabulary, which is used to describe classes, instances and their relations in an unambiguous way. In addition, class specification, instance inclusion and value restriction in ontology can be specified in a strict class hierarchy. Similar to knowledge bases, ontologies provide both intention and extension types of knowledge, where the intention is about class or generic information that describes the particular domain, while the extension is about instances, i.e. the specific instantiations of the domain description. Ontologies enhance the functionality of the Web in many ways, such as, consistency checking, interoperability support, data or schema validation and verification, configuration support, structured search, exploiting generalization and specialization information. Interested readers can refer to [24].

The OWL Web Ontology Language (OWL[23]) is the most expressive of the ontology languages currently developed for the Semantic Web. OWL has three different levels of language: OWL Lite, OWL DL(for description logic), and OWL Full. These three sub-languages are in increasingly expressive in order. Beyond the RDFS [2] which defines classes, subclasses, properties and subproperites, OWL defined property restrictions. Both intentional and extensional knowledge can be represented in OWL. In order to represent class information and data-type information, OWL defines class constructs such as disjointWith and supports logic set theory such as intersectionOf,

unionOf, complementOf. Additionally, the universal quantifier is specified by allValueFrom as a restriction on corresponding property, which means that for each instance of the class or data type restricted, all values for the corresponding property must belong to that instance. On the other hand, the existing quantifier, is specified by someValueFrom as a restriction on corresponding property, which means that for each instance of the class or data type restricted, at least one value for the corresponding property belongs to the instance. An OWL document is composed of RDF [1] triples, but those triples have been assigned the specific semantics. In this way, OWL provides a semantic interpretation for those RDF graphs. OWL is a Description Logic markup language for defining Web ontology. Thus, there is a mapping of constructors (Table 2.1) and axioms (Table 2.2) in OWL which correspond to the Description Logics constructors and axioms:

OWL Constructor	DL Syntax	Example
intersectionOf	$C_1 \sqcap \cdots \sqcap C_n$	$Woman \sqcap Mother$
unionOf	$C_1 \sqcup \cdots \sqcup C_n$	$Man \sqcup Woman$
complementOf	$\neg C_1$	$\neg Male$
oneOf	$\{p_1, \ldots, p_n\}$	$\{sue, \ldots, mary\}$
allValueFrom	$\forall \mathcal{P}.\mathcal{C}$	$\forall hasChild.Student$
someValueFrom	$\exists \mathcal{P}.\mathcal{C}$	$\exists hasChild.President$
minCardinalityQ	$\geq n\mathcal{P}.\mathcal{C}$	≥ 1 $hasChild.Student$
maxCardinalityQ	$\leq n\mathcal{P}.\mathcal{C}$	≤ 3 $hasChild.Student$

Table 2.1: OWL Constructs and Their Corresponding DL Syntax

OWL is a Web Ontology language which is formally defined to provide explicit specification of certain domains shared between large groups of stakeholders. When Web contents are defined by ontologies, machines can achieve a certain degree of understanding of the data. Thus web ontologies are ideal to enable automatic interoperation between entities on the Web. As far as Semantic Web Service is concerned, the interoperation may include: service matching, service composition, service planning, service exception handling etc.

OWL Axiom	DL Syntax	Example
subClassOf	$C_1 \sqsubseteq C_2$	$Student \sqsubseteq Male \sqcap Female$
equivalentClass	$C_1 \equiv C_n$	$Man \equiv Human \sqcap Male$
disjointWith	$C_1 \sqsubseteq \neg C_2$	$Male \sqsubseteq \neg Female$
sameIndividualAs	$p_1 = p_2$	$zl07 = Zheng\ L$
differentFrom	$p_1 \neq p_2$	$zl07 \neq dr36$
subPropertyOf	$\mathcal{P}_1 \sqsubseteq \mathcal{P}_2$	$hasSon \sqsubseteq hasChild$
equivalentProperty	$\mathcal{P}_1 \equiv \mathcal{P}_2$	$Cost \equiv Price$
inverseOf	$\mathcal{P}_1 \equiv \mathcal{P}_2^-$	$hasChild \equiv hasParent^-$
transitiveProperty	$\mathcal{P}_1^+ \sqsubseteq \mathcal{P}_2$	$Ancestor^+ \sqsubseteq Ancestor$

Table 2.2: OWL Axioms and Their Corresponding DL Syntax

OWL-S [4] is developed, as an OWL-based Web Service Ontology, and aims to support tools and agent technology for the automation of services on the Semantic Web. "OWL-S supplies Web Service providers with a core set of markup language constructs for describing the properties and capabilities of their Web Services in unambiguous, computer-intepretable form. OWL-S markup of Web Services will facilitate the automation of Web Service tasks including automated Web Service discovery, execution, interoperation, composition and execution monitoring". OWL-S supports both simple and complex interactions of Web Services, the primary motivation in defining OWL-S is as follows:

- Automatic service discovery: this means the act of automatically locating a machine-interpretable description of a Web Service which matches a given set of functional, non-functional or other constraints. The goal is to find an appropriate Web Service resource by using an automated process. For example, to find a weather forecast service for a city where service requester is located.

- Automatic service invocation: this means that when an appropriate service has been located, without further interaction required, the identified services are executed automatically by a computer program or agent. For example, when a weather forecast service is found, the response is automatically sent to the service

requester.

- Automatic service composition and interoperation: this means that based on high-level description of a service requirement, automatically composing several Web Services into a new Web Service. To support this processes, multiple tasks may be involved, such as automatic service selection, composition, and interoperation.

- Automatic service execution monitoring: the execution transactions that rely on Web Services are vulnerable to the problems of those Web Services. OWL-S offers the ability to find out where in the process the request is and whether any unanticipated glitches have appeared by providing declarative descriptors for the state of execution of services.

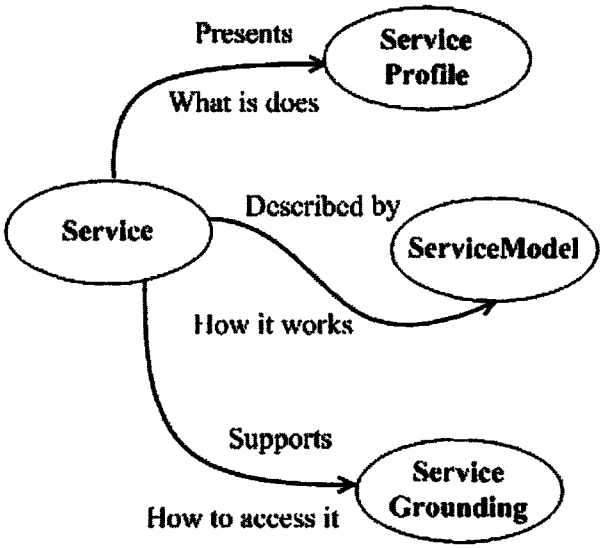

Figure 2.2: Top Level OWL-S Class Hierarchy

Industry has focused on manageability and modularization of Web Services, while academic research has been concerned more with expressiveness of service descriptions [69]. As an upper ontology for services, OWL-S defines four basic classes, namely Service, ServiceProfile, ServiceModel and ServiceGrounding, where the Service is at the top level of service description hierarchy, and it is an organizational point of reference for

any Web Service. Having been defined as the properties of a class Service, ServiceProfile, ServiceModel and ServiceGrounding represent different perspectives of a Web Service, and each of these properties provides an essential type of information about the service respectively (See Fig 2.2).

ServiceProfile: this description document answers the question of *what a particular service does.* The ServiceProfile represents a comprehensive description of information necessary for any service application to decide if this particular service matches the requirement. In addition to the functional description, the ServiceProfile also includes non-functional requirement descriptions such as limitation on service applicability, quality of service and the constraints which a service requester must satisfy. The relation between class Service and ServiceProfile has the form of "Presents", which means an instance of Service presents a ServiceProfile description.

ServiceModel: this description document answers the question of *how is a particular service used.* The ServiceModel offers details about: conditions applied on using a service, what are the steps that lead to service outcomes etc. This description is useful for further analysis of the service matching, composition of existing service descriptions, coordination among multiple services involved and monitoring the execution of the service. The relationship between class Service and ServiceModel has the form of "DescribedBy", which means an instance of Service is described by a ServiceModel description.

ServiceGrounding: this description document answers the question of *how a particular service could be accessed.* Generally, the ServiceGrounding specifies which communication protocols, message formats are used. The relation between class Service and ServiceGrounding has the form of "Supports", which means an instance of Service have a support property referring to a ServiceGrounding.

Briefly, the ServiceProfile forms the description used for service discovery, the ServiceModel groups with ServiceGrounding to form the description used for execution of a service.

In current OWL-S, a service is intentionally modeled as a process which is a specification of how to interact with that service, but not a program to be executed. The processes can be categorized as:

- **An atomic process** which only performs single function, i.e. return one message in response. There are no sub-processes or further executions in an atomic process. Usually, it is directly invocable, with a single message receiving and a single message return pattern.

- **A composite process** which performs multiple functions by aggregating multiple existing atomic processes and needs to maintain some states during a service transaction. A composite process can be decomposed into either atomic or composite sub-processes. To specify the method of decomposition, OWL-S defines the control constructs, such as sequence and iterate. However, what a process specifies is not what behaviors a service will do, instead, it specifies that to achieve overall service effect, what behaviors the client should do to invoke every sub-process.

- **A simple process** which is not invocable, but is regarded as having one step execution. A simple process in OWL-S is out of two purposes, first it may be used to provide a view of an atomic process, in this case, the simple process is realized by the atomic process; second, it may be used represent a simplified version of composite process, in this case, the simple process expandsTo the composite process. Unlike an atomic process and a composite process, a simple process is not related to any grounding, because it is uninvocable.

Each service has input and output parameters. Besides the input and output parameters, the functional description is described by two sets of conditions, namely precondition and effect, where precondition must be true for the process to be executed and effect represents the conditions that must be true immediately after the service execution completes. For instance, if we are going to use a car rental service, a

precondition would be that the credit card is valid, and an effect of the execution of the service would be that the credit card is charged a certain amount (namely, the price of the rental fee). The four elements (input, output, precondition, effect) of service functional description are referred as IOPE.

In the the service ontology hierarchy structure, inputs and outputs are the subclasses of parameter. Input to output represents a process of data transformation produced by a service execution. The input required by an atomic service must come from the service requesters, while the input required by composite services could be either directly from the service requesters or from output generated by previous steps. Furthermore, a service can have as many inputs and outputs as required, also including none. Specified by inputs and outputs, data transformation formed the good foundation for reasoning about service syntactic inconsistencies which has been defined in classical software literatures [70, 65]. For example, two processes are type compatible, if the output parameterType process A is a subtype of the input parameterType of process B.

The last two elements in IOPE are represented as an object property called expression, which connect processes to their precondition and effect. Precondition to effect represents a transition of the world state, when a service execution completes. If a process has a precondition, to execute the process properly, the precondition must be true. Effects are the changes of world states, when execution of a service completes. However, the evaluation of precondition can generate results other than true or false, for example, being believed to be true, being known to be true or unknown. If the precondition is evaluated as false, the results of performing the process are undefined. The combination of effect and output constructs the service result data structure. The term "result" is used to refer to a coupled service output and effect. Effects represent the condition changing of the world, while outputs represent the message passing. Finally, a service can have as many preconditions and effects as required, including none.

From the studies of the current Web Service descriptions from both industry and

academic research, we can tell that the two efforts have different foci and have proposed two different solutions to this new computing paradigm. The service description languages proposed by industry define a stateless client-server model of synchronous or uncorrelated asynchronous interactions [6, 21]. On the other hand, the current Semantic Web Service Ontology language (OWl-S) is layered on top of these low-level communication layers and provides a richer semantic description of Web Services. In summary, WSDL can be viewed as a low-level communication language and protocols which define *how* to access a Web Service, while OWL-S aims at providing an answer to the questions, such as:

- *why* one uses a certain Web Service.
- *what* this Web Service actually does [73].

2.3 Web Service Composition as Planning

The Semantic Web is an approach to making the Web resources machine interpretable. To achieve the goal of automatic Web Service composition, Semantic Web Service Language (OWL-S) is developed to specify the various aspects about the semantics of services, which has been proved to be useful for various intelligent service behaviors such as service discovery, description and composition. Recently, the planning techniques which result from the AI research discipline have been employed to automate the composition of Web Services. The planning technique can be viewed as a problem solver which helps us to "know what to do before things are done". In classical planning representation, the planning system is defined by the initial state of the world, a set of operators which correspond to actions changing the current state, and a goal condition which is a set of ground formulas. The state is a set of ground formulas expressed in first order language. During the planning process, the planning agent attempts to find a sequence of operators or actions which transform the initial world state into the goal state, i.e. a model which satisfies a given set of goal formulas. A state is a complete

view of the world, a ground formula p holds in the current state S iff $p \in \mathcal{S}$. The description of each action is specified by:

- the precondition which specifies what formulas should belong to the current state in order for the action to be applicable.
- the add list which denotes the positive effects of an action.
- the delete list which denotes the negative formulas that may no longer be true and therefore must be deleted.

In the rest of this work, we will use the symbol $\models$ to represent logical entail. Typically, an action θ can be represented as a triple $\theta = (name(\theta), pre(\theta), effect(\theta))$. An action θ is applicable in a state $\mathcal{S}$ when the preconditions p are satisfied in the state, i.e. $\mathcal{S} \models pre(\theta)$. Traditionally, planners represent the world state with a relational database and thus precondition evaluation is very fast. Applying the effects of an operator is completed by adding or deleting entries from the database. If the goal condition $\mathcal{G}$ is a set of formulas with variables, $\mathcal{S}$ satisfies $\mathcal{G}$ is denoted by $\mathcal{S} \models \mathcal{G}$. During the planning process, a resolution theorem prover is adopted for the operator precondition evaluation, for the goal formula validation in the last world state, and also for directing the search.

In classical planning, a domain theory denotes a description of the possible actions, which are specified in some formal language and may be executed. Usually, domain theories follow some state-transition model. A transition system is a tuple $T = \langle \mathcal{A}, \mathcal{S}, \mathcal{S}_0, \mathcal{R}, \mathcal{S}_g \rangle$ where $\mathcal{A}$ is the set of actions which may change the world model from one state to another, $\mathcal{S}$ is the set of all possible states of the world, $\mathcal{S}_0$ is the set of initial states and $\mathcal{S}_0 \subseteq \mathcal{S}$, $\mathcal{R}$ is the transition relation and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$, and finally, $\mathcal{S}_g$ is the set of final states.

Description logics and Semantic Web Language provide the logic foundations for the service ontology language OWL-S, and this formalization provides the formal semantics of the services, i.e. a domain theory. Similarly, the state change produced by

the execution of the service is specified through the service functional properties: precondition and effect. Precondition presents logical conditions that should be satisfied prior to the service being requested. Effects are the result of the successful execution of a service. Because of this similarity, the majority of the methods of service composition as planning use OWL-S as the service description language.

Over time, planning has become a rich research area in the AI research discipline, and many AI planning techniques have been developed to support different levels of expressivity and solve different planning problems. This has resulted in a wide range of different formats and notations, thus the semantics of domains have often been ambiguous. To address this problem, the Planning Domain Definition Language (PDDL[36]) was developed to serve as a standard planning domain and problem specification language and is widely recognized as a standardized input for various planners. Moreover, the development of OWL-S has been strongly influenced by PDDL language. It is straightforward to map from one language representation to another. Because OWL-S descriptions could be easily translated into the PDDL syntax, then different planners could be exploited for further service synthesis.

Among the approaches of service composition as planning is the HTN planning system SHOP2, which is a Hierarchical Task Network (HTN) planner. In [71], the SHOP2 planner is applied for automatic service composition and provides a sound and complete algorithm to translate OWL-S service descriptions to a SHOP2 domain. Motivated by the task decomposition in HTN planning, HTN planner is adopted to solve the semantic service composition problem. The authors also claim that the HTN planner is more efficient than other planning languages, such as Golog. Finally, the proposed system is also capable of executing information-providing Web Services during the planning process, which means the proposed system is suitable for the service planning even with an incomplete initial state of the world.

[46] proposed a Golog-based approach for the automatic composition of Web Services. Golog is a logic programming language built on top of the situation calculus.

The goal of this work is to provide a semantic web agent the programming capability of writing generic procedures for service-based tasks. The knowledge state of a planning agent can provide a logical encoding of the Semantic Web Service Descriptions in the language of the situation calculus. The general idea of this method is that various service requests are predefined as generic procedures. In addition, with user's specified constraints, these generic procedures can be customized at runtime to goal instances. Moreover, Services are defined as two subclasses, namely, PrimitiveServices and ComplexServices. Primitive services are similar to atomic service in OWL-S, which denotes a service that expects one message and returns one message in response. ComplexServices are compositions of multiple services. Finally, using procedural programming language control constructs (if-then-else, while, and so forth), a composite service becomes a set of atomic services.

2.4 Inconsistency in Software Engineering and Default logic

Web Services are emerging application technologies to reuse software as the service cross the web. The Web Service model provides a powerful access channel to integrating services across multiple applications inside and outside the enterprise to achieve a business objective. At the same time, it also creates a new set of challenges other than opportunities. A critical issue with the current Web Service's model is the problem of how to establish consistent service composition. The composition of existing Web Services from multiple independent parties entails inconsistency problems, and the difficulties arise as Web Services are dynamically composed.

When Web Services execute in an open-ended environment, uncertainties can easily lead to conflicts. Here the service composition can be viewed as: the chain of the world state transitions produced by the execution of a single business capability, unions of

specific domain properties from all distinct parties involved and application of general infra-structure rules and constraints. Since a Web Service's implementation is a software module, it shares many similarities with traditional software systems [77, 78]. In terms of software engineering, inconsistence denotes "any situation in which a set of descriptions does not obey some relationship that should hold between them "[49]. It can also be described as "a state in which two or more overlapping elements of different software models make assertions about the aspects of the system they describe which are not jointly satisfiable "[70]. Generally speaking, inconsistency means situations where a given set of requirements cannot be simultaneously satisfied. As far as the software development life cycle is concerned, these inconsistencies can arise in system requirements, design specifications and, quite often, in the descriptions that form the final implemented software product [50]. In terms of the software requirement engineering, the inconsistency can been broadly classified as [65]:

- **Syntactic Inconsistency**: this is caused by terminology inconsistency or improper grammar.
- **Semantic Inconsistency**: this is about conceptual meanings.

The inconsistency classification above can be easily applied to Semantic Web Service Descriptions. As mentioned in the previous section, the functional property of a service is specified by the input, output, precondition and effect. Clearly, the combination of input and output can be used for syntactic inconsistency handling, while both precondition and effect are logic expressions which can be used for the semantic inconsistency handling, as shown on Table 2.3

However, in this work, what our concerns is that the set of assertions about a given service composition are inconsistent with respect to the domain theories, in other words, there is no way to satisfy all these assertions together. We will explain our ideas in more detail in later chapters. Having reviewed a number of research surveys in the software engineering literature, a wide range of inconsistencies can arise during requirements

Property	Range	Kind	Handling Inconsistency
hasInput	Input	Parameter	Syntactic
hasOutput	Output	Parameter	Syntactic
hasPrecondition	Condition	Expression	Semantic
hasEffect	Expression	Expression	Semantic

Table 2.3: IOPE Used to Handle Inconsistencies

engineering because of the following reasons:

- Multiple perspectives: System requirements are elicited from multiple stakeholders, who may have distinct and often contradictory viewpoints on the requirements of the proposed system.
- Non-functional Vs functional requirement: non-functional requirement or quality factors often contradict to functional requirements. Distinct non-functional requirements can even contradict each other, for example, security requirement may contradict to the system accessibility requirement.
- Requirement evolution: during the process of system evolution, modification of the some elements leads two actions occur, namely adding and deleting, these two actions may cause the inconsistency to other existing elements.

In traditional software engineering, normally the system specification is under the control of a single stakeholder who, at least in principle, determines a consistent set of requirements [37]. This process makes it easier to handle inconsistencies caused by various sources. On the other hand, service composition is characterized by multiple-stakeholder environments and distributed deployment and dynamic execution. Consequently, the inconsistencies in the context of service composition may result from:

- limited control and knowledge about the multiple application domains involved.
- distributed responsibilities, accountability, authority, ownership, and control etc.
- lack of coordination and collaboration among those autonomous software agents.

Clearly, incompleteness of information and uncertainty can easily result from such an open environment. By an environment, or service composition environment, we refer to that the part of the real world with which various services are to interact, including the people and organizations as well as the implementation of the software modules, and hardware devices. To achieve consistent service composition, it is critical that the behavior of services has the ability to reason about and adapt to a changing environment. When the services are dynamically composed, possibly, the candidate services from different parties have conflicting goals or perspectives for the underlying service composition to be built. The application of default rules [67, 63, 44] to the state of knowledge during the service composition process is beneficial for this practical purpose. Default logics allow for the representation of contradictory beliefs within the same model, and then they could highlight potential conflicts and possible ways of resolving them. Thus default logic representations may ideally model the various aspects of the service composition problem, specially, when it is not particularly clear what the underlying service composition to be. By adopting the default rules, each service node in the distributed environment is allowed to routinely make assumptions about the permanence of objects and the typical features or properties of objects. Following the process of service composition, corrections to the assumptions also can be smoothly accommodated. If a contradiction is detected, the old conclusions are discarded to incorporate new knowledge.

Default logic is introduced by Reiter in [63]. It provides formal principles for making inferences when the information at hand is incomplete and for retracting of the previous conclusions when contradictory information appears. A default theory is a pair (D, W) where W is a set of first-order formulae representing the facts which are known to be true with certainty and D is a set of defaults. A default δ has the form of

$$\frac{\varphi : \psi_1, \ldots, \psi_n}{\chi}$$

where φ, $\psi_1, \ldots, \psi_n$ and χ are classical predicate logic formulas. The formulas φ are

called the prerequisite and are denoted by $pre(\delta)$, formulas $\psi_1, \ldots, \psi_n$ are called the consistency conditions or justifications and are denoted by $just(\delta)$, and formulas χ are the consequence of the default and are denoted by $cons(\delta)$. Note that the formulas in a default must be ground. On the other hand, defaults with free variables are called open defaults, which are usually interpreted as the schema representing a set of defaults.

Intuitively, given a default $\varphi : \psi_1, \ldots, \psi_n / \chi$, its informal meaning is the following: if φ is true and it is consistent to assume $\psi_1, \ldots, \psi_n$, then we can conclude χ. To formalize this interpretation, for a given default $cons(\delta)$, we should know in which context φ must be true and with what $\psi_1, \ldots, \psi_n$ should be consistent. Only if the consistency of the set of justifications has been tested against the set of known facts, can the defaults be subsequently applied. Given $\delta = \varphi : \psi_1, \ldots, \psi_n / \chi$ is applicable to a deductive closed set of formulas E, if and only if $\varphi \in E$ and $\neg\psi_1 \notin E, \ldots, \neg\psi_n \notin E$. A default theory introduces so-called extensions which represent possible consistent interpretations of the available information, i.e. possible world views based on the given default theories. One default theory can have more than one extension, which is the way of the default theory to allow for the representation of contradictory beliefs within the same model. E is an extension of (D, W) if and only if E is a deductively closed set satisfying the following properties:

- An extension E should include W which is the set of facts containing the information available, i.e. $W \subseteq E$.
- An extension E should be deductively closed, which allows us to perform classical logical reasoning as well as to draw conclusions based on the default theory.
- All defaults that are applicable with respect to E have been applied, which means that the extensions E is maximal possible world views.

The extension of the defaults has been defined in [76]. Let (D, W) be a default theory, then E is an extension of (D, W) if and only if $W \subseteq E$ and E is deductively

closed where

$$D_E = \{\frac{pre(\delta) : just(\delta)}{cons(\delta)} \mid \delta \in D, \psi \in just(\delta), \neg\psi \notin E\}$$

An operational definition of default extensions has been given in [76, 12]. Let a given default theory be $T = (W, D)$, $\Pi = (\delta_1, \ldots, \delta_n)$ is a sequence of defaults from D, in other words, Π denotes a possible order in which the set of defaults are applied from D. With the sequence Π, there are two sets of first-order formulas, namely, $In(\Pi)$ and $Out(\Pi)$, where

1. $In(\Pi)$ is represented by $W \cup \{cons(\delta_1), \ldots, cons(\delta_n)\}$ for $i \in \{1, \ldots, n\}$. In addition, $W \cup \{cons(\delta_1), \ldots, cons(\delta_{i-1})\} \models pre(\delta_i)$. As usual $\models$ here denotes the classical consequence relation. Thus, $In(\Pi)$ represents the current knowledge base after the default δ_n has been applied.

2. $Out(\Pi) = \{\neg\psi | \psi \in just(\delta)$ for some δ occurring in $\Pi\}$. Contrary to $In(\Pi)$, $Out(\Pi)$ contains formulas that should not turn out to be true, which means that $Out(\Pi)$ represents the knowledge that should not be contained in the current knowledge base after subsequent application of other defaults **OR** the current knowledge base will contain the contradiction.

A key property of intelligence whether exhibited by man or by machine is flexibility. This flexibility is intimately connected with the defeasible nature of commonsense inference. We are all capable of drawing conclusions, acting on them, and then retracting them if necessary in the face of new evidence. If our computer programs are to act intelligently, they will need to be similarly flexible [31]. Default logics allow one to make plausible conjectures when faced with incomplete information about the problem at hand. Default reasoning [62] denotes the process of arriving at the conclusion based upon patterns of inference of the form "In the absence of any information to the contrary, assume...". Default logics support reasoning with incomplete information based on assumptions, thus it provides means for adapting to a changing environment.

Service composition is about implementing new value-added services, whose application logic involves the invocation of operations offered by other services. The new service is a composite service, and the invoked services are the components. Typically, the components of a composite service are provided by different parties and an existing application needs to be exposed over a network for use by unknown requesters. Consequently, the information required by any given dynamic service composition could be ambiguous, inconsistent and incomplete. To clarify the ambiguity, resolve the inconsistencies and accommodate the incompleteness of information, we adopt an explicit notion of defaults in current semantic service description, and this added default notion makes service composition more flexible in the following ways:

- During the process of service composition, particularly in the face of incomplete information or uncertainty, it is possible to draw tentative conclusions based upon an incompletely specified initial set of knowledge and to act on them.
- It also provides an adequate account of how composing a value-added service progresses as a consequence of new information being added or existing conclusions being retracted.

Adopting an explicit notion of defaults is an attempt to make precise statement about the intended behavior of a service composition and its environment, which can help achieve a higher degree of flexibility. As well as achieving a higher degree of flexibility, it is also important to assure the consistency of service composition. To assure consistency, the service composition should be viewed as a continual process of re-validation and re-verification. Obviously, the various defaults involved in any given service composition need to be handled in an appropriate way. In this case, the default principle must be applied which allows for the representation of contradictory beliefs within a partially specified service composition, and then potential conflicts could be highlighted and possible ways of resolving these conflicts could be identified.

2.5 Related Works

In the last section, we have briefly introduced the concept of inconsistency in software engineering and conducted a basic review about default logics. The use of default rules in software development is beneficial for practical purposes, in particular, in the discipline of requirement engineering [11, 79]. Software development usually starts with system requirement acquisition from multiple stakeholders who are involved in the process of system development. However, these stakeholders may have conflicting requirements for the system to be developed. Because default logic allows for the representation of conflicting requirements within the same model, default logic representations may ideally model these potentially conflicting requirements.

Among different approaches, paper [45] explored the possibility of automated support for detecting software requirement inconsistencies. In this work, it provided a practical way to combine nonmonotonic logic and rapid prototyping to help maintain software. According to this paper, the changes of the software system environment introduce the inconsistencies. For this reason, the specifications are classified as immutable or mutable. Mutable specifications are statements about the software and its environment which remain the same for all time. A mutable specification is a statement which is believed or assumed to be true, i.e. default assumptions. Default assumptions here are characterized as: typically being true, but not true in all situations. Based on an extension to logic programming, this work also presents a standard mechanism for handling exceptions caused by default assumptions. By extending to logic programming, this proposed Computer Aided Prototyping System presents an improved automated capability for detecting the inconsistencies introduced by software changing.

Since the system environment as well as stakeholder requirements specifications change, i.e. new requirements may be added and existing ones may be deleted, requirement evolution is an important source of contradiction among the requirements. Every

phase of software development is characterized by continued evolution. Paper [79] presented a logic framework for modeling and reasoning about requirement evolution in the construction of information systems. In this proposed framework, a requirement model serves as a basis for reasoning with and about requirements. A set of operations are defined to provide a formal basis for requirement evolution. At a meta-level, the requirements model which may include incomplete and inconsistent requirements, is viewed as a nonmonotonic theory, specially a default theory. Suppose that the requirement model is represented in some formal language with a well-defined semantics, requirement evolution operations can map between theories of this meta-level logic. In this approach, the operations provide a formal basis for requirements evolution, and incompleteness of information and inconsistent requirements can be captured and handled by a rich meta level logic. By adopting the default principle in requirement management, it is possible to obtain complete requirements models by taking initially incomplete requirement specification and applying relevant defaults from the domain. Beyond this, corresponding to multiple possible extensions of the corresponding default theory, the formal mechanism of acquiring default extension was provided to resolve the contradiction and select from amongst multiple possible views of a requirements model.

The works we mentioned in this subsection represent approaches to reasoning about incompleteness of information and inconsistencies in software requirement engineering by adopting the default principles. These approaches also provide a useful starting point for defining semantically well founded system for managing a changing environment during the system development life cycle. Resolving requirement conflicts caused by the changing environment and consistently combining reusable components are often tasks which have to be solved for both traditional software system development and service composition. In fact, the notions and ideas presented in constructing software systems are highly applicable to service composition, as service composition could simply be viewed as a type of composition of independent reusable software components

at runtime. Examples for such scenarios are service composition for B2B systems with a large set of business partners. Each of these business partners operates its service applications according to its own goals and priorities. When these services are deployed, each of them has more or less limited knowledge of each other. Because of the distributed environment and the ignorance of one another, it is almost impossible to have a central explicitly coordinated consistent model of dynamic service composition at all times. Thus the approaches used to help the traditional software system development may not be appropriate to solve the problem of dynamic service composition. In this work, based on our proposed extensions to OWL-S, we use the planning techniques to resolve the conflicts and unexpected misbehaviors introduced by the incompleteness of information and inconsistency during the process of dynamic service composition, which will be explained in detail in the next two chapters.

In the research area of Semantic Web Service, there is another outstanding research effort, the WSMO working group [7]. WSMO [8] aims to further the development of Semantic Web Services by working towards further standardization in the area of SWS languages, and through the development and implementation of a common architecture and platform for Semantic Web Service. WSMO is based on four concepts: Web Services, ontologies, goals and mediators.

Similar to OWL-S, ontologies in WSMO provide machine-readable semantics for the information used by all actors implied in the process of Web services usage, either providers or requesters, allowing interoperability and information interchange among components.

The capability of a Web Service is described in terms of precondition, postcondition, assumption and effect. One of contributions we claimed in this work is to extend the current Semantic Web Service Description OWL-S by introducing the concept of service assumptions which allow reasoning with incomplete information. Note that the concept of service assumption we proposed is different from the assumption defined in WSMO, because:

- The concept of service assumption proposed in this work is an extension to current version of OWL-S, and most importantly, is based on the default theory, but assumption defined in WSMO does not explicitly state that it is based on the default theory. In other words, our proposed service assumption has explicit purpose of reasoning with incomplete information and adapting to a changing environment.

- To make the service assumption more flexible and more accurately describe the problem of service composition, we further classify the service assumption into different categories (See Chapter 3).

The mediators are another core part of WSMO, which allow the linking of heterogeneous resources and the resolution of incompatibilities that arise at data, protocol and process levels. The current version of the WSMO specification distinguishes four types of mediators: ooMediators, ggMediators, wgMediators and wwMediators:

- OO Mediators

  1. Connect ontologies to any other components, including mediators.
  2. Resolve mismatches and conflicts between ontologies.

- WW Mediators

  1. Link Web Services to services they depend on.
  2. Resolve representation differences through OO Mediators.

- WG Mediators which links Goals and Web Services.

- GG Mediators which connect generic and refined Goals.

Generally speaking, WSMO is designed to become a standard and it represents one of the most comprehensive frameworks for Service Oriented Architecture, which includes:

- Goals which describe some state that a user may want to achieve.
- Ontologies which are the formal specification of the knowledge domain used by both the web service to express its capability, and by the goal to express the desired world state.
- Mediators which are used to solve different interoperability problems.

2.6 Summary

The importance of Web Services has been recognized and widely accepted by industry and academic research. In this chapter, we have conducted a brief review with respect to both industry and academic research. Generally speaking, industry has focused on modularization of different service layers, while academic research has emphasized expressiveness of service descriptions. To tackle the problem of the consistent service composition, we have also surveyed relevant literature in software engineering. To give the service the ability to adapt to a changing environment during service composition, we have also given a brief background to default logic which supports reasoning with incomplete information. [39] accurately described software engineering as a discipline of description. To improve the current Semantic Web Service Description for the purpose of consistent service composition, in next chapter, we will extend the current Semantic Web Service Description by introducing the concept of "Service Assumption".

Chapter 3

Extending OWL-S by Service Assumption

3.1 Atomic Services

The platform neutral nature of Web Services creates the opportunity for building composite services by dynamically combining the functionalities of existing atomic or complex services. An atomic service is a directly invocable computer program which has no subprocesses and can be executed in a single step. Usually, an atomic service is invoked by a request message sent by a client. After processing the request, the atomic service produces a single response to the client. Also, an atomic service can be combined with other atomic or composite services to create value-added services. One of characteristics of atomic services is that there is no ongoing interaction between the client and the service, while typically a composite service needs to maintain some state.

In the context of Semantic Web Services, the functional description of the service is expressed in terms of information transformation and state change. Information transformation is represented by input and output properties, while a state change produced by the execution of a service is specified through the precondition and effect properties. The functional description of an atomic service in OWL-S is represented as $\langle I, O, P, E \rangle$ where:

- I is the input property that describes the information a service requires to proceed with the computation.
- O is the output property that describes what information a service returns back

to the client.

- P is logical condition that should be satisfied prior to a service being requested. Precondition is an optional description of an atomic service. In other words, with an atomic service, if no precondition is specified, then the service is always executable. On the other hand, if any precondition is specified, then the service cannot be performed successfully unless the precondition is satisfied.

- E represents the effect of an atomic service, and lists the changes that the service execution imposes on the current state of the world. However, effect is fundamentally different from the output in that effect describes conditions in the world, while output describes information.

It is worthwhile mentioning that there is another commonly used term, called: postcondition which has some similarities to the term "effect". Generally speaking, postcondition is a condition or predicate that must always be true just after the execution of an action or after an operation in a formal specification. There are also some differences between these two terms. Firstly, OWL-S is an ontology of services which draws upon well-established work in a variety of fields and includes work in AI on standardizations of planning language PDDL. The effect is the part of PDDL action definition. Secondly, from the definition we can tell that postcondition is about the condition being true or false after the execution of an action, while on the other hand, effect is about changing the world state after the execution of an action. Although, there is difference between these two terms, effect depends on conditions that hold true of the world state at the time the action is performed.

Checking the consistency of information production in the service composition has been proposed by [78] in terms of the compatibility and conformance of the input and output, which is also can be found mostly in the theory of programming languages. In this proposed work, we restrict attention to another dimension, aiming to tackle the problem of conflicts in service composition which may be caused by conflicting

perceptions, assumptions and goals of the multiple services involved in the service composition process.

3.2 The Need for Service Assumptions

Web Services build on emerging best practices of eliminating programming module dependencies from business logic. The basic motivation of service oriented computing is to allow a high degree of flexibility to create the value-added composite service in a dynamic fashion. Web Service composition shares many similarities with traditional component-based software system. They both provide aggregated functionality via reassembling various existing objects, and emphasize [53] same design principles such as reusability, replaceability, flexibility and extensibility. As [34] observes: the notions and ideas presented in constructing software components are highly applicable to Web Services, as they could simply be viewed as a type of software component architecture but with the addition of yielding a standard communication model. Briefly speaking, Web Service composition model aims to connect the functional units of web applications as services through well-defined interfaces.

Obviously, Web Services are provided by a large number of independent parties. Often, these independent parties do not necessarily share the same objectives and background. [37] has pointed out that requirements engineering has traditionally assumed that the system to be designed is under the control of a single stakeholder who (at least in principle) determines a consistent set of requirements. Modern distributed systems, however, do not fit this mold, so requirements engineering must adapt to handle them. A multi-stakeholder distributed system (MSDS) is a distributed system in which subsets of the nodes are designed, owned, or operated by distinct stakeholders. The nodes of the system may, therefore, be designed or operated

- in ignorance of one another
- or with different, possibly conflicting goals.

The fundamental goal for service-oriented computing is to connect business functions across the Web both between enterprises and within enterprises. Clearly, Web Service are running in a distributed environment, and the ignorance of one another may result in incompleteness and uncertainty of the information during the process of service composition. Making a decision upon incomplete or uncertain information easily fails to achieve consistency. Hence, to achieve reliable service composition, it is critical for Web Services to have the ability to adapt to a changing environment. As a result of this adaptability, it is possible to reason about a changing environment and to deal with the exceptions resulting from the incompleteness and uncertainty of the information. The focus of this work is restricted to bridging the gap between the semantic service descriptions (See Section 2.2) and multiple operational domains involved and to maintain the consistency of service composition.

Traditional software development usually starts with system requirement acquisition from multiple stakeholders, however, these stakeholders may have different perspectives about the system. Following the acquisition step, analysis needs to be carried out to detect and resolve conflicts between those different viewpoints. Some approaches have been proposed in requirement engineering research disciplines [22, 43, 60]. Unlike traditional software development, during the process of service composition, often, it is impossible to have a clear cut boundary, based on which the potential conflicts can be detected and resolved. Unpredictable service executions and a dynamically changing environment complicate dynamic service composition in many ways.

OWL Web Ontology Language for services specification (OWL-S [4]) leverages the rich expressive power of OWL [23] together with its well-defined semantics to provide richer descriptions of Web Services. Service ontologies can be used to map service functional descriptions and domain properties into a standardized logic [14] so that they can be machine understandable and interpretable. Recently, Semantic Web Rule Language (SWRL) [38, 15] has been proposed to define service process preconditions and effects, process control conditions and their contingent relationships in OWL-S.

Though OWL-S is endowed with more expressive power and reasoning options when combined with SWRL, the description provided by a combination of OWL-S and SWRL about service composition is still only a partial picture of the real world. Most of what we know about the world, when formalized, will yield an incomplete theory precisely because we cannot know everything - there are gaps in our knowledge [62]. Similarly, the ontology of services, is finite and incomplete. Thus, a service composition specified by OWL-S has to deal with partial or incomplete knowledge. Currently, OWL-S has no mechanism for handling incomplete knowledge during the process of dynamic service composition. Inspired by [63, 44, 67], in this work, we propose service assumption which aims to bridge the gap between the semantic service descriptions and multiple operational domains involved in dynamic service composition.

3.3 Service Assumptions

The concept of service assumption extends the functional service description defined by the current semantic Web Service ontology OWL-S [4]. The proposed service assumption will supply service providers with an option for describing the properties and capabilities of their Web Services in a more precise way. Service assumptions can be used to define a collection of default conditions regarding service policies, where each assumption is believed in lack of evidence to the contrary, and is taken to be true until the contrary is proved. For instance, the assumption made by a car rental service could be "city car does not run on dune". The service assumptions are believed when information is incomplete, but these assumptions also can be revised over time to incorporate new knowledge. Because of the heterogeneous nature of the Web Service execution environment, incomplete information in the process of service composition may occur either because of the unavailability of certain information or to keep the formulation simple at the start. Service assumptions here allow reasoning with incomplete information by the default settings and then revising conclusions ever made to reflect

new information about the problem. By extending the current OWL Web Ontology Language for Services (OWL-S) specification, an atomic service ws_i in this proposed work is described by a tuple $\langle p_i, e_i, a_i \rangle$, where

- p_i is a set of sentences representing the precondition that must be true for the atomic service to execute, which can be represented as: $p_i = \{p_i^1, \ldots, p_i^n\}$. In addition, we define the each sentence in $\{p_i^1, \ldots, p_i^n\}$ as a primitive precondition.
- e_i is a set of sentences representing the change of world state including both positive and negative effects, which can be represented as: $e_i = \{e_i^1, \ldots, e_i^n\}$. In addition, we define the each sentence in $\{e_i^1, \ldots, e_i^n\}$ as a primitive effect.
- a_i is a set of sentences representing service assumption, which can be represented as: $a_i = \{a_i^1, \ldots, a_i^n\}$. In addition, we define the each sentence in $\{a_i^1, \ldots, a_i^n\}$ as a primitive assumption.

Given a service $ws_i = \langle p_i, e_i, a_i, \rangle$, informally, its semantics can be interpreted as: if p_i can be satisfied, and if it is consistent to assume a_i, then we may conclude that e_i can be applied. Note that p_i and a_i are different, because

- It must be possible to establish that p_i is true for ws_i to be invoked. On the other hand, we only need to establish that a_i is consistent with what is known, i.e. nothing is known that contradicts a_i.
- p_i is a strong condition which **must** be true in order to execute the service ws_i, while a_i is a weak condition. Initially we assume a_i to be true, unless we get additional information which is explicitly contradictory to a_i.
- Precondition is logical condition that must be satisfied before a service is executed, which means that after satisfaction of this precondition, this precondition no longer affects the succeeding executions in the service composition. On the other hand, service assumption has the character of being persistent, which means

that after service execution, normally, service assumption still has some impact upon the underlying service composition. Here, precondition can be viewed as the conditions regarding the eligibility of a service to be used, while the service assumption can be viewed as the conditions regarding the way in which a service is used.

OWL-S is a formal language which aims to provide precise and rich declarative specification of a wide variety of properties about Web Services in order to support automation of a broad spectrum of activities across the Web Service life cycle such as discovery, selection, composition, negotiation and contracting, invocation and monitoring of progress. In the current version of OWL-S, the vast majority of techniques focus on the modeling and specification of the Web Service alone. Certainly, the declarative specifications of prerequisites, consequences of application of individual services and data flow interactions need to be defined precisely and related to each other. These have been defined by means of $\langle I, O, P, E \rangle$ in the current version OWL-S. However, currently OWL-S lacks support for reasoning about the composite service made up of existing atomic or complex services and their environment. Insufficient service assumptions about the changing environment and uncertainty can easily leads to incomplete or inaccurate composite service specifications. Thus, in parallel, the assumptions made about incomplete knowledge and uncertainty also need to be made explicit and documented. The general goal of adding service assumption as a part of service specification is to allow reasoning about and adaption to a changing environment in the process of service composition. This might be seen as

1. accurately describing the service composition environment, in which most instances of a concept generally have some property, but not always.

2. representing the hypothetical guesses about the incompleteness and uncertainty.

3. some combination of each.

3.4 Service Assumption as Functional Properties

Currently, there is no way for OWL-S to describe the various assumptions about the multiple independent application domains involved in service composition. In addition, there is no mechanism to guarantee that the service execution has the anticipated effects when there is insufficient knowledge available during service composition execution. By adopting service assumptions into OWL-S, we can conduct reasoning about what is known in the composite service execution context against various domain assumptions. Thus the ontology for Web Service becomes more complete and closer to the real world. To use the service assumption in unified manner with other service properties specified by current Semantic Web Services Language OWL-S, together with input, output, precondition and effect, hasAssumption in this proposed work is also defined as one of the service's functional properties, which would allows various assumptions to be captured and recorded in readily accessible fashion. The syntax is proposed as follows:

```
<owl:Class rdf:ID="Assumption">
<owl:subClassOf rdf:resource="&expr;#Expression"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="hasAssumption">
<rdfs:domain rdf:resource="#Process"/>
<rdfs:range rdf:resource="&expr;#Condition"/>
</owl:ObjectProperty>
```

Code Segment 3.4: Extending OWL-S with Service Assumptions

Like the functional property hasPreconditioin which has been already defined in OWL-S, hasAssumption is also represented as logical expressions and denotes conditions that are evaluated with respect to the service composition environment. Expressions here are represented by any allowed logical language. To be consistent with the current OWL-S specification, we choose the Semantic Web Rule Language (SWRL)[15, 38] to represent service assumptions. The construct of hasAssumption here is used to specify one of the assumptions of the service and ranges over an assumption instance. After

adding service assumptions, a Web Service is connected to its functional properties shown in Table 3.1:

Domain	Property	Range	Kind
Service as Process	hasAssumption	Condition	Expression
Service as Process	hasPrecondition	Condition	Expression
Service as Process	hasEffect	Expression	Expression

Table 3.1: Connection between Web Services and Their Functional Properties

3.5 Service Assumptions about Individuals

In a service composition context, service assumptions can be viewed as a hypothesis. However, this hypothesis is about individuals rather than the terminology, where terminology is about how concepts (classes) or roles (properties) are related to each other in a given application domain and individuals are instances of these classes or properties. Terminology represents the characteristics of the world, for instance, MasterCard is always subclass of Credit Card, while the facts about individuals represent our current state of knowledge that may change over time, for instance, a particular MasterCard may have expired. The reason to exclude the usage of terminology as the assumption is intuitive, because the terminology in ontologies is used to model the world as we know it.

Moreover, the proposed service assumption can represent two types of different knowledge in the service composition context. Let x, y, z denote either a variable, an OWL individual or an OWL data value, $\mathcal{C}$ denote an OWL class description and $\mathcal{P}$ denote OWL property, then we have:

1. Concept assumptions $\mathcal{C}(x)$, which asserts x is an instance of the OWL class description $\mathcal{C}$.

2. Property assumptions $\mathcal{P}(y, z)$, which asserts z is value of the OWL property $\mathcal{P}$

for y.

The proposed extensions to the current OWL-S make it possible to capture the various assumptions of the service domain. We also propose to use SWRL expressions in OWL-S assumptions, thus we can use the expressive power of rules to facilitate service conflict reasoning. Here, we give examples to show a simple case of service assumption. The example is taken from the car rental service, which has the policy "the rented city car cannot drive on certain road conditions", and this policy is enforced by the service assumption. The example is as follows:

```
<process:hasAssumption>
<expr:SWRL-Condition rdf:ID="DriveCarInProperWay">
<rdfs:label>notDriveOn(car, roadCondition) & notDriveOn(car,
   anotherRoadCondition)
</rdfs:label>
<rdfs:comment>Typically this condition should also include more road
   conditons the car cannot drive on,
to keep this example simple, all other details are left out for this
   example.
</rdfs:comment>
<expr:expressionBody rdf:parseType="Literal">
<swrl:AtomList>
<rdf:first>
<swrl:IndividualPropertyAtom>
<swrl:propertyPredicate rdf:resource="#NotDriveOn" />
<swrl:argument1 rdf:resource="#cityCar" />
<owlx:Individual owlx:name="Dune" />
</swrl:IndividualPropertyAtom>
</rdf:first>
<rdf:rest>
<swrl:AtomList>
<rdf:first>
<swrl:IndividualPropertyAtom>
<swrl:propertyPredicate rdf:resource="#NotDriveOn" />
<swrl:argument1 rdf:resource="#cityCar" />
<owlx:Individual owlx:name="Unsealed_Road" />
</swrl:IndividualPropertyAtom>
</rdf:first>
<rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil
   " />
</swrl:AtomList>
```

```
</rdf:rest>
</swrl:AtomList>
</expr:expressionBody>
</expr:SWRL-Condition>
</process:hasPrecondition>
```

Code Segment 3.5: Example of Using Service Assumption

The example is written by using Semantic Web Rule Language (SWRL) syntax which extends the abstract syntax of OWL-S described in the OWL Semantics. Unfortunately, rules written in SWRL are not particularly human-readable. Thus the example is provided here to explain this abstract syntax. Generally, a service assumption is represented as a rule, which has the form: *antecedent* $\Rightarrow$ *consequent*, where the symbol $\Rightarrow$ denotes the logical "imply" and both antecedent and consequent are generally defined as conjunctions of atoms, having the form of $\psi_1 \wedge \cdots \wedge \psi_n$. Using this syntax, a rule states that the composition of "city car not drive on dune" and "city car not drive on unsealed road" properties implies the "DriveCarInProperWay" property would be written:

$$\neg DriveOn(?cityCar, dune) \wedge \neg DriveOn(?cityCar, unsealedRoad) \Rightarrow DriveCarInProperWay(?cityCar)$$

In the example above, the antecedent of the rule consists of two primitive assumptions, i.e. $\neg DriveOn(?cityCar, dune)$ and $\neg DriveOn(?cityCar, unsealedRoad)$, and the consequent of the rule is $DriveCarInProperWay(?cityCar)$. Since we have defined that service assumptions can only contain OWL individuals, possibly with variables, an assumption expression becomes equivalent to a conjunctive query. Informally, the example can be explained as: if both "city car may not drive on dune" and "city car may not drive on unsealed road" are consistent with what is known in the context of the service composition, then it is assumed that the car will drive in the proper way, where consistent means without the information to the contrary. In this example, the contrary information will be "city car drives on dune" or "city car drives on unsealed road".

3.6 Classification of Service Assumptions

3.6.1 Hard Assumptions and Soft Assumptions

Results from the study of default logics serve as a basis for understanding service assumptions. Default logics provide formalisms to deal with assumptions or beliefs. Generally speaking, default logics perform the retraction of beliefs when new information is presented which contradicts those beliefs. To use service assumptions for the real world application in more flexible way, there are two distinct usages of service assumptions which need to be taken into consideration. The first case is the restriction about the usage of a Web Service, while the second case makes assumptions to provide warning information, aiming to ensure that service requester gets a satisfactory result. Informally, the classification of service assumptions as follows:

1. **Hard Assumptions**: this kind of assumption is used to strengthen the service policy. In the context of the service composition, the hard assumption cannot be violated or the service composition will fail.

2. **Soft Assumptions**: this kind of assumption is used for the purpose of providing warning information. The soft assumption only states that, typically, most instances of a service composition have some property.

The example provided in last section uses a hard assumption, which aims to enforce the service policy - car usage. In other words, if there is conflicting information against the car usage assumption in the context of the service composition, then this car rental service will not become a piece of the generated composite service. The second kinds of service assumption is much like the first, in that it is considered to be a conflict in service composition, if conflicting information appears. Unlike the first kind of service assumption, in face of conflict information, soft assumption will provide the warning information, but the service requester has the option of how to deal with that. In other words, the service requester could choose either to ignore this conflict information or

to discard the chosen Web Service which produced the conflicting information. Using a soft assumption is quite normal in our real life. For example, a tobacco vendor receives a request for tobacco from a customer. The tobacco vendors assume that the consumer of tobacco knows that "Smoking Causes Heart Disease", and usually this warning information is displayed on the tobacco pack. However, if face of this warning information, the customer still can make his own decision. Returning to our car rental service, one example of using a soft assumption might be that, a particular car model normally is rented as a wedding courtesy car, but a service requester wants to rent this car for a long distance trip. Renting this car may be very costly, so the service provider kindly provides such warning information. However, the service requester can make his decision whether or not to rent this car in face of the warning information.

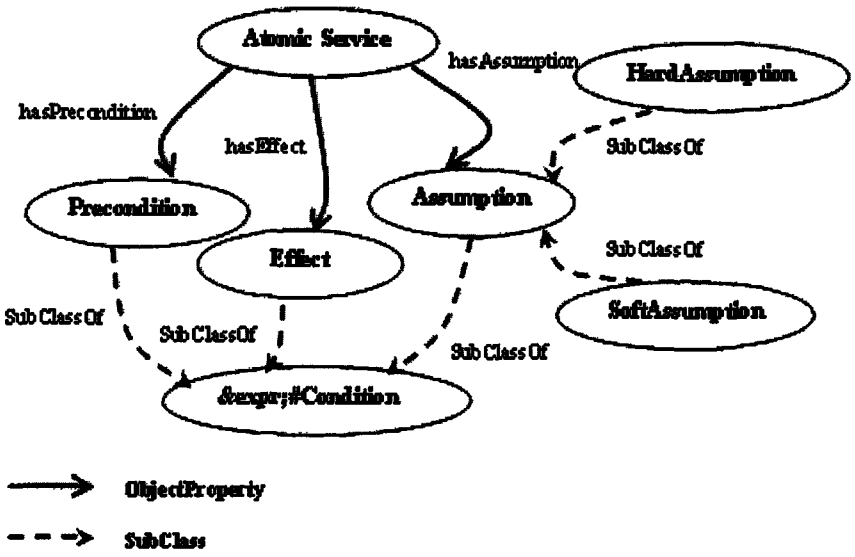

Figure 3.1: Extended Atomic Service Description

After the classification of soft assumptions and hard assumptions, the complete view of atomic service description is shown on Fig 3.1. The following code segment shows how HardAssumption and SoftAssumption are represented in OWL syntax as class axioms:

```
<owl:Class rdf:ID="HardAssumption">
```

```
  <rdfs:subClassOf rdf:resource="#Assumption"/>
</owl:Class>

<owl:Class rdf:ID="SoftAssumption">
  <rdfs:subClassOf rdf:resource="#Assumption"/>
</owl:Class>

<owl:Class rdf:ID="Assumption">
  <rdfs:comment> The most general class of Assumption </rdfs:comment>
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#HardAssumption"/>
    <owl:Class rdf:about="#SoftAssumption"/>
  </owl:unionOf>
</owl:Class>

<rdf:Description rdf:about="#HardAssumption">
  <owl:disjointWith rdf:resource="#SoftAssumption"/>
</rdf:Description>
```

Code Segment 3.6.1.1: Class Axioms of Hard/Soft Assumption

All ontologies specified by OWL-S are written in OWL. OWL classes are formally described using Description Logic [14] that precisely defines the requirements for membership of the class. OWL classes are interpreted as sets that contain individuals. Furthermore, classes may be organized into a superclass-subclass hierarchy. Subclasses specialize their superclasses. Following the specification for a service assumption in OWL-S we have defined in Section 3.3, the code segment 3.6.1.1 above is composed of four small blocks of code, which describe the hierarchical structure of the classification of a service assumption. The first block states that the class HardAssumption is a specialization of the class Assumption, in other words, the set denoted by class HardAssumption is a subset of class Assumption, i.e. HardAssumption $\sqsubseteq$ Assumption. Similarly, the second block states that the class SoftAssumption is also a specialization of the class Assumption, i.e. SoftAssumption $\sqsubseteq$ Assumption. The third block says that the class Assumption has been created as the union of class HardAssumption and SoftAssumption, i.e. Assumption $\equiv$ SoftAssumption $\sqcup$ HardAssumption. Finally, HardAssumption and SoftAssumption are disjoint from each other so that an individual cannot be a member of

more than one of them, i.e. SoftAssumption ⊓ HardAssumption ⊑ ⊥. The corresponding OWL abstract syntax and Description Logic syntax are summarized in Table 3.2:

OWL Abstract Syntax	DL Syntax
SubClassOf (HardAssumption, Assumption)	HardAssumption ⊑ Assumption
SubClassOf (SoftAssumption, Assumption)	SoftAssumption ⊑ Assumption
UnionOf (SoftAssumption, HardAssumption)	SoftAssumption ⊔ HardAssumption
DisjointClasses (SoftAssumption, HardAssumption)	SoftAssumption ⊓ HardAssumption ⊑ ⊥

Table 3.2: Class Axioms of Hard Assumptions and Soft Assumptions

After defining class axioms of HardAssumption and SoftAssumption, the following code segment shows that how hasHardAssumption and hasSoftAssumption, as property axioms, are related to the service functional property hasAssumption:

```
<owl:ObjectProperty rdf:ID="hasSoftAssumption">
  <rdfs:subPropertyOf rdf:resource="#hasAssumption" />
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="&expr;#Condition"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasHardAssumption">
  <rdfs:subPropertyOf rdf:resource="#hasAssumption" />
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="&expr;#Condition"/>
</owl:ObjectProperty>
```

Code Segment 3.6.1.2: Defining Sub-Properties of Service Assumptions

The code segment 3.6.1.2 above defines the inheritance hierarchical structure over hasAssumption. This structure describes two specific functional properties, namely, hasHardAssumption and hasSoftAssumption. Intuitively, the connection between a service and hasAssumption is inherited by these two subproperties. The corresponding OWL Abstract Syntax and Description Logic Syntax are summarized in Table 3.3:

OWL Abstract Syntax	DL Syntax
SubPropertyOf (hasHardAssumption, hasAssumption)	hasHardAssumption $\sqsubseteq$ hasAssumption
SubPropertyOf (hasSoftAssumption, hasAssumption)	hasSoftAssumption $\sqsubseteq$ hasAssumption

Table 3.3: Property Axioms of hasHardAssumption and hasSoftAssumption

3.6.2 Transient Assumptions and Persistent Assumptions

In this sub-section, we will present an example which demonstrates the need to classify the service assumptions based on the relation between a service assumption and its associated service effect. In Section 3.3, a Web Service ws_i has been defined as: $ws_i = \langle p_i, e_i, a_i \rangle$. Here, let a_i^i be a primitive assumption, which is a sentence in a_i, i.e. $a_i^i \in a_i$. Let $\{\neg e_i^1, \ldots, \neg e_i^n\}$ denote the negation of a service effect e_i. Because both service assumption and effect are generally defined as conjunctions of atoms, if a_i^i is also a sentence in $\{\neg e_i^1, \ldots, \neg e_i^n\}$, i.e. $a_i^i \in \{\neg e_i^1, \ldots, \neg e_i^n\}$, then clearly, $a_i \cup \{\neg e_i^1, \ldots, \neg e_i^n\} \neq \emptyset$, in other words, a contradiction can be inferred from ws_i itself, i.e. $a_i \cup e_i \models \bot$. In this case, the service assumption a_i of service ws_i contradicts its associated service effect e_i. One might hold the view that such a service represents nonsense. However, considering the following real world example

```
(: Web Service AAA Shopping Online Member Reg
 :parameters  (?appl - Applicant
                    ...)
:precondition ( and (?appl  OlderThan 18)
                ...)
:effect (?appl isAAAmember)
:assumption (?appl isNotAAAmember))
```

Code Segment 3.6.2: Shopping Online Member Registration Service

For the description of the sample service above, we use syntax similar to that of the Planning Domain Definition Language [36]). The example is about AAA Shopping Online Member Registration Service, the precondition of the service is that "the applicant must be older than 18", the effect is that "the applicant is a member" and the

assumption is that "the applicant is not a current member". One interpretation is that as long as the fact of the applicant older than 18 years old can be proved, and so far there is no known evidence that the applicant is a current member, then after applying this service, the applicant is a member. The service assumption in this example is used to prevent the same applicant having two memberships. However, this simple service also can be interpreted in another way: as long as the fact that the applicant is older than 18 years can be proved, and assuming that the applicant may never be a member (even after the service), then after applying this service, the applicant is a member. The second interpretation makes nonsense of this sample service description, because the service description itself is self-defeating. Since this service assumption explicitly contradicts its associated service effect, the issue will be the lifespan of the service assumption.

As stated earlier, service assumptions represent hypothetical guess that is believed in the lack of evidence to the contrary, thus it also acts as one of the consistency conditions needs to be tested under specific contexts during the process of service composition. Typically, the consistency condition has to be met both before and after the effect of service is applied. However, to avoid the self-defeating problem in the real application of Web Services, based on the relation between service assumption and its associated service effect, we classify service assumptions as: (also See Table 3.4)

1. **Transient Assumptions**: for any Web Service ws_i, if a contradiction can be inferred from the union of a service assumption a_i and its associated effect e_i, i.e. $a_i \cup e_i \models \bot$, then we refer to a_i as the *transient assumption.* When a transient service assumption plays the role of being the consistency condition in a service composition context, this condition only needs to be tested before the effect e_i of given service ws_i is applied, but **not after.**

2. **Persistent Assumptions**: for any Web Service ws_i, if there is no contradiction inferred from the union of a service assumption a_i and its associated effect e_i,

i.e. $a_i \cup e_i \not\models \perp$, then we refer to a_i as the *persistent assumption*. When a persistent service assumption plays the role of being the consistency condition, this condition has to be tested both **before** and **after** the effect e_i of given service ws_i is applied.

Classification	Union of a_i and e_i	Checking Methods
Transient Assumption	$a_i \cup e_i \models \perp$	Before Applying e_i, not After
Persistent Assumption	$a_i \cup e_i \not\models \perp$	Before & After Applying e_i

Table 3.4: Transient Assumptions Vs. Persistent Assumptions

Note that unlike the soft assumptions and hard assumptions, which are classified by service providers at the service design time and used to specify the service requirements in a flexible way, transient assumptions and persistent assumptions are classified by the reasoner at service composition planning or execution time and used to avoid ill-formed service descriptions. In addition, it is still possible for a persistent assumption to be outdated in the process of service composition. We will explain the concept of outdated assumption in next chapter. The classification of transient assumptions and persistent assumptions does not need to be explicitly specified in OWL-S, as these concepts are only used by the service composition reasoner at execution time, not service providers at design time. For the different categories of service assumption options available at different phases, please see Table 3.5.

Phase	Service Assumption Available
Service Design Time	SoftAssumption OR HardAssumption
Service Execution Time Or Composition Planning Time	Persistent HardAssumption OR Persistent SoftAssumption OR Transient HardAssumption OR Transient SoftAssumption

Table 3.5: Service Assumption Options at Different Phases

Although we have proposed different ways to use service assumptions, it is still

necessary that service providers make a service specification in a sensible way. After all, if a sentence is an ill-formed sentence, we do not blame the language in which it is written. Clearly, the application of a service requires that a consistency condition is satisfied. What makes meeting that condition complicated is the fact that services can interact in complex ways. To create a value-added service composition in a distributed environment, it is usually the case that there are multiple independent parties involved in this process. In next chapter, we will explain the reasoning process with proposed service assumptions for service composition.

3.7 Summary

Service composition is created in a distributed environment, in which typically, there are multiple independent parties involved in the process. Often, these independent parties do not necessarily share the same objectives and perspectives, thus various services may be designed or operated in ignorance of one another. As a result, we must make assumptions about things we don't specifically know. Default attributes are a powerful kind of knowledge, since they permit useful conclusions to be made...[74].

To reason about and adapt to a changing environment, in this chapter, we have extended the OWL-S to a richer service description representation schema by introducing service assumptions. This aims to bridge the gap between the semantic service descriptions and multiple operational domains. The goal of adopting service assumptions into the service functional description is to enable service applications:

- to be more flexible and intelligent. The flexibility and intelligence result from default service assumptions which have the defeasible nature of commonsense inference. When a service composition is executed in a heterogeneous environment, by adopting service assumptions, it is possible to draw tentative conclusions and smoothly accommodate corrections to the assumptions.

- to be executed in a consistent manner. Because of distributed responsibilities,

accountability, authority, ownership and control, in the process of service composition, sometimes the action must be taken in the presence of incomplete knowledge or uncertainties. The proposed service assumption attempts to make precise statements about the intended behavior of the service and its environment. The more accurate and precise service the description of the problem, the more reliable the decisions we make.

Also in this chapter, we have explained the semantics of the service assumption, and proposed a method to define a service assumption as a functional property. To adopt the service assumption in more flexible way, we also further classify the service assumption into two categories (See Fig 3.1): soft assumptions and hard assumptions. Furthermore, to avoid the self-defeating service description, based on the relation between a service assumption and its associated service effect, we classify the service assumption as transient assumption and persistent assumption, which is summarized in Table 3.4.

In next chapter, together with the proposed service assumptions, the knowledge based planning framework will be developed, which will attempt to tackle the problem of incompleteness of information and uncertainties in service composition context.

Chapter 4

Service Composition Framework

4.1 Preliminaries

In the previous chapter, we have extended current OWL-S by introducing service assumptions, aimed at handling incomplete information and uncertainties which may cause conflicts in a service composition context. It is often assumed that a business process or application is associated with some explicit business goal definition that can guide a planning-based composition tool to select the right service [46, 72]. Planning is a complex problem which has often been investigated in the AI literature [30, 59, 28, 40, 75]. [66] characterize the problem of planning as follows : "Planning can be interpreted as a kind of problem solving, where an agent uses its beliefs about available actions and their consequences, in order to identify a solution over an abstract set of possible plans".

In this chapter, we attempt to provide a formal framework for reasoning about incomplete knowledge during the service composition planning process by adopting service assumptions. This proposed framework is layered on top of Semantic Markup for Web Services (OWL-S [4]) and Semantic Web Rule Language (SWRL [38]). Service assumptions, as the extension to current OWL-S, are represented by default literals which describe possible incomplete information or uncertainties in service composition planning domains. During the service composition planning process, when knowledge is

insufficient, the proposed service assumptions enable us to incorporate intelligent decision making in terms of default principles [63, 67]. In the process of service composition planning, it is impossible to have a clear cut boundary between multiple application domains and it is usually infeasible to have a complete view of the world. Hence, the transitions during a service composition planning process are described in terms of between different **states of knowledge** rather than between different states of the world. In this work, our conflict checking and state transitions use a set of structured rule conditions which govern the state transitions and guide the derivation of the new state. Given that a specification of a service composition which is associated with some explicit goals can be generated via transition-based planning, state transition conditions are defined to describe all permitted behaviors in a service composition context. The proposed framework exhibits a number of characteristics. Among them, it is possible to make tentative conclusions when the information available is insufficient, also the conclusion that has been made can be revised over time to incorporate new knowledge. Before we proceed to further describe our service composition framework, we will first introduce the definitions of the service selection function and the composite services.

4.1.1 Service Selection

Unlike software component compositions, the automated process of service compositions holds some additional critical issues, such as service matching, selection and retrieval (See Fig 4.1). UDDI [5, 41] provides a mechanism for the Web-Wide Service registry, in which descriptions of Web Services are stored and searched by category. OWL-S allows us to semantically describe the capabilities of Web Services, thus it is possible to perform logical inferences for the service matching. [51] Provides one way to combine these two techniques, so that services defined in OWL-S can be registered with UDDI in a way that allows UDDI engines to exploit OWL-S semantic information to facilitate the retrieval of Web Services. In this proposed framework,

- ws_i represents an atomic service.

- WS is the set of all Web Services, $ws_i \in WS$.
- All Web Service descriptions are held in their corresponding categories $\{cat_1, \ldots, cat_n\}$. cat_i is a tangible area split from the service registry, for example downloadable Multimedia.
- CAT is the set of all service categories where

$$cat_i \in CAT, cat_i \in WS, cat_i = \{ws_1, \ldots, ws_m\}$$

- Service selection function $sel : CAT \rightarrow WS$ which takes a certain service category as its input and gives us an atomic service based on the service matching i.e. $sel(cat_i) = ws$.

To reason about and adapt to a changing environment in the process of service composition, we have extended current OWL-S by introducing the concept of service assumption. The proposed service assumption is defined as one of atomic service's functional properties. Certainly, a composite service can be built by combining existing atomic or some other predefined composite services, however, any composite service is composed of multiple atomic services indeed, that's why we defined an atomic service as the output of our service selection function. Every atomic service in the rest of this chapter refers to the Web Service which is produced by the service selection function defined above. For more details about the service matching, interested readers may refer to [51, 52].

4.1.2 Composite Service

Intuitively, a composite service $CompWS$ which performs combined functions may include multiple atomic services. $CompWS$ is the combination of the multiple atomic services ws_i, where $0 < i < n$. A composite service $CompWS$ can be represented as:

$$CompWS = \{sel(cat_1), \ldots, sel(cat_n)\}$$

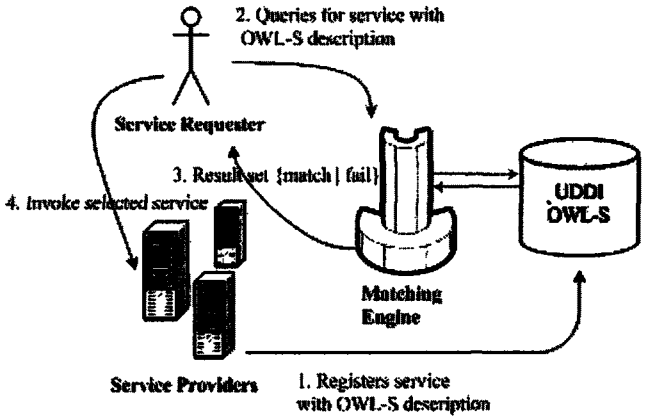

Figure 4.1: Service Selection

Participants in the service composition do not necessarily share the same objectives and background, so, without a mechanism to reason about incomplete knowledge and its side effects during service execution, conflicts easily arise in the service composition context. The underlying idea of this planning framework is to generate a consistent service composition plan by applying the default principles [63, 67].

4.1.3 Service Composition Planning Domain

A planning domain represents the service composition problem space, which is defined by the states of the domain, the available service descriptions used to simulate actions, the assumptions made in the planning process and the state transitions caused by the execution of services. In this framework, a service composition planning domain $\mathcal{D}$ is the tuple $\langle \mathcal{S}, \mathcal{WS}, \mathcal{M}, \mathcal{TR} \rangle$ where:

- $\mathcal{S}$ is a finite set of states.
- $\mathcal{WS}$ is a finite set of available services as actions.
- $\mathcal{M}$ is an assumption database.
- $\mathcal{TR}$ is a 2 - tuple $\langle \mathcal{T}, \mathcal{R} \rangle$ where $\mathcal{T}$ is the state transition function and $\mathcal{R}$ is the sequence of rule conditions related to each state transition.

4.1.4 Service Composition Planning Problem

Briefly, a service composition planning problem $\mathcal{P}$ is the problem of finding a path or constructing a process of actions given the planning domain initial and goal states. In addition, this path must not contain inconsistent problem states. Service composition planning problem $\mathcal{P}$ for a planning domain $\mathcal{D} = \langle \mathcal{S}, \mathcal{WS}, \mathcal{M}, \mathcal{TR} \rangle$ is 4-tuple $\langle \mathcal{D}, S_0, S_g, \mathcal{G} \rangle$, where

- $\mathcal{G}$ is the goal of the service composition to achieve.
- $S_0 \in S$ is the initial state.
- S_g is any state such that $S_g \models \mathcal{G}$.

A goal $\mathcal{G}$ is set of conjunctions of atoms which need to hold in a desired world state or final state. A service composition plan for a goal is a sequence of state transitions of atomic services, and the transitions lead from an initial state to a final state where all ground atomic formulas in the goal are true. A solution to a service composition planning problem is to jointly compose some selected abstract Web Services as actions to get a certain task done i.e. achieve the goals specified by the service requester. Compared with the traditional planning techniques, our primary interest here is to provide an improved capability for detecting certain kinds of inconsistencies which result from incompleteness of information or uncertainties. This enhanced planning framework is endowed with an ability to reason about and adapt to a changing environment.

4.2 Service Composition as Planning

In this section, we will present all the main features of our service composition framework. We proceed incrementally by further decomposing the service composition domain. In the previous section, the service composition domain was defined as:

$$\mathcal{D} = \langle \mathcal{S}, \mathcal{WS}, \mathcal{M}, \mathcal{TR} \rangle$$

We begin by looking at the state of knowledge $\mathcal{S}$.

4.2.1 State of Knowledge

$\mathcal{S}$ is a finite or recursively enumerable set of states, i.e. $\mathcal{S} = \{S_1, \ldots, S_n\}$. A state S_i in this work is extensionally defined as a set of positive or negative ground atomic formulas. As usual, we refer to atomic formulas as the formulas which are function-free and do not contain variables, for example, the formula $Registered(john, surfClub)$ might represent a state in the problem of Surf Club Online-Registration service. However, the formulas such as $Registered(brother(john), surfClub)$ is not allowed. Also in this framework, there is no delete list. If *john* canceled his registration, instead of adding $Registered(john, SurfClub)$ to the delete list, we simply change the formula as $\neg Registered(john, SurfClub)$. Those formulas which may change their values during the state transition are called fluent, while those which do not change are called state invariant. A state of knowledge during the process of service composition is characterized by the truth values of some combination of fluents and invariants, that is, predicates describing relevant properties of the domain of discourse, where every fluent necessarily is either true or false.

We are motivated by the problem of building composite service by dynamically composing the functionalities of existing atomic or complex services. Often, Web Services are provided by independent parties. In real world service composition domains, it is impossible to have complete information about the entire world. So, we emphasize that states in this framework represent states of knowledge rather than states of the world. Incompleteness of information and uncertainties arise when information regarding a domain is not explicitly represented in the context. In the course of the service composition, the composition planner or system will have incomplete information or uncertainties about the world, because of

- a desire to keep the formulation simple at the start.

- the unavailability of certain information.
- the high cost involved in obtaining certain information.
- the abundance of irrelevant data.
- a lack of understanding.

The task of the service composition planning for the planning agent is to find some composition of available services that transforms its given initial knowledge state into one that satisfies the goal conditions. Traditional planning techniques [30, 18] simply assumes that there is an initial world state which could provides a complete description about the world. To solve the problem of dynamic service composition, this simplifying assumption is unrealistic. As the set of services grows very large and all services are located in service registries in a distributed way across the Web, obviously trying to complete the initial state will be practically impossible in the real world problem domain. Given the nature of Web Services, we cannot assume that any planner or system knows the all information needed to build a service composition before the planning process starts. In this work, the initial state is just one of states of knowledge defined by the domain theory, which does not specify all knowledge relevant to the planning task. The initial state is either an empty set or is a proper subset of the initial world model. For the reasons we list above, in this work, the service composition framework is designed for planning domains in which the information about the initial state of the world may not be complete, but some of this information can be acquired through planning time sensing operations [29, 32, 17].

To better understand the sensing operation, first we need to explain the terms on-line and off-line planning. Classical planning is done off-line. An off-line planner generates a complete plan before the task is performed, which means that the plans are generated prior to execution and then the generated plan is fed to the on-line execution module. On the other hand, an on-line planner typically interleaves planning and execution.

Second, it is necessary to distinguish atomic services as actions that change the state of the world from those that only return information. The former are well known in planning literature, and classical planning languages [30, 36] can only represent this type of actions, i.e. actions with causal effects, for example, the execution of an action causes the sentence φ to become true. The latter do not actually affect the world during planning, serving only the sole purpose of information-providing, i.e. only changing the knowledge that the planning agent has. However, gathering certain information from information-providing services makes it necessary to execute them at plan time. To cope with incomplete information during the process of service composition planning, the planner is endowed with the capability of on-line execution of information-providing services and off-line simulation of world altering services. In the context of service composition, the planning process starts with an incomplete initial state and executes sensing actions for information gathering purpose, which add new knowledge to the state (See Fig 4.2).

Besides the service matching, one typical use case of the information gathering during the service composition process is precondition evaluation of the selected services. For instance, when requesting a flight-booking service, consider the precondition of the service shown below:

```
(:Web Service Flight-Booking
:parameters (?fn-flightNo ?d-requiredDate ?p-price
             ?cc-CreditCardNo ?lmt-limit)
:precondition (and (?fn isAvailable ?d)
                   (?cc isValidCreditCard)
                   (?p < ?lmt))
:effect ...)
```

Code Segment 4.2: Precondition of A Flight-Booking Service

The simple flight-booking service above saying that there must be more tickets available for a given required date, the credit card must be valid and the available limit on the credit card must be higher than the price of the air ticket. After the relevant

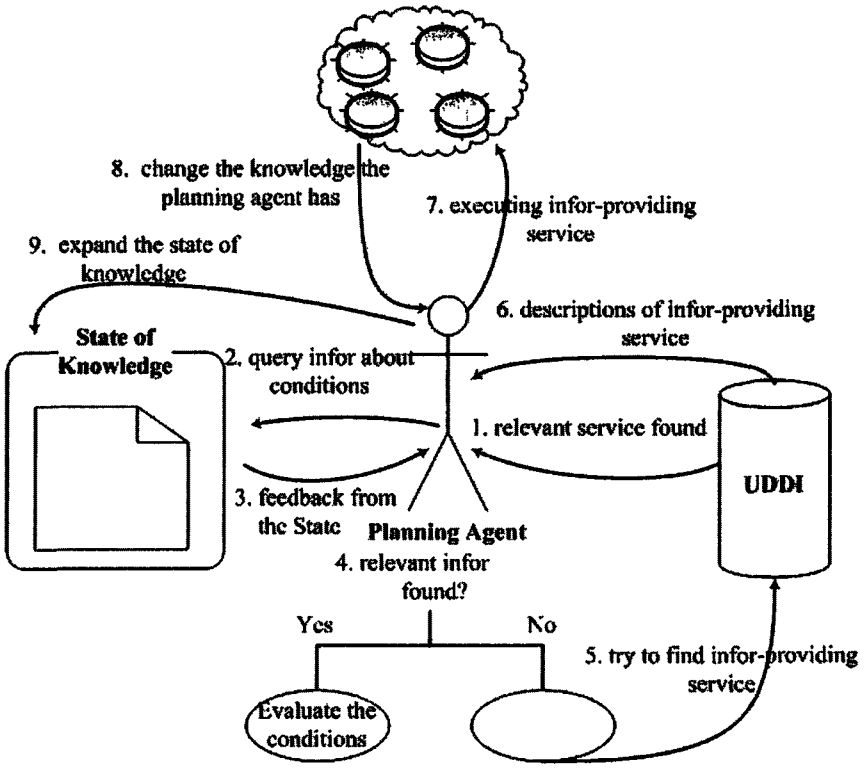

Figure 4.2: Sensing Operations in Service Composition Planning

service is found in UDDI, the planning agent must first evaluate the precondition by querying the state of knowledge described above. If relevant information is explicitly represented in the state of knowledge, then the variables for instance ?flightNo and ?date etc bind to actual values. After the variable binding, the planning agent can compare these values and verify that the precondition holds. As not all the information relevant to a service composition problem may already be known at the initial state or during the course of the planning process, sometimes it will be necessary to perform plan-time sensing operations to acquire such kind of information. .For example, to acquire the information about the availability of a ticket, the planning agent may need to query external information-providing services which could access the flight center's database. Similarly, to validate the credit card, the planning agent may need to query the status of the credit card authorization.

Atomic services which only provide information are fundamentally different from those which change the state of the world. In general, the planning agent will execute information-providing atomic services at various points of time, while world-altering services will never be executed at plan-time, and the effect of world-altering services will be simulated in off-line mode. However, adding the sensing operation for creating a service composition plan in a heterogeneous environment is not a trivial activity, since it requires coordination with various existing information resources which may involve problems of currency control, privacy and cost etc. To simplify the case of sensing operations, we make some assumptions:

- incomplete but correct information is available about the world state.
- information providing services are executable in the initial state.
- information gathered from these services cannot be changed by external services in the planning process.

These assumptions are certainly very restrictive, for this reason, some solutions are proposed, for example, the same information-providing service is prohibited from executing more than once. Also some other approaches are suggested, for example, interleaving planning with execution. However, this is not the problem we are trying to solve in this work, thus the topic is out of the scope of this paper. For advanced techniques for information gathering during service composition planning, interested readers may refer to [33, 42].

In the knowledge representation literature [68, 56], incomplete of the knowledge has been classified as: either absence or uncertainty. Adopting this classification about the incomplete knowledge in the service composition planning process, we refer to the missing facts as the absence of information. On the other hand, uncertainty is the subjective measure of certainty about service interactions, which may be caused by ignorance of one another when multiple independent parties are involved in the process. Clearly, uncertainty and absence are essentially different, thus we use different techniques to handle these two distinct types of incomplete information. The absence of information is handled by the sensing operation mentioned above. However, what makes dynamic service composition complicated is the fact that, during the process, services interact in complex ways. In this work, one of our contributions is to extend OWL-S by introducing service assumption. The proposed service assumption can be used to describe the service composition environment which may be not specifically known. As a consequence of this more precise description of the service composition environment, it is possible for us to deal with exceptions and resolve the inconsistencies which are caused by uncertainty. From now on, we will concentrate on the service composition consistency problem, which may be caused by uncertainty during the process of interactions among multiple independent parities.

In the following subsection, we will introduce the assumption database $\mathcal{M}$ which stores and manipulates various assumptions during the process of service composition planning. In turn, the values of various maintained service assumptions may influence

on the evolution of the service composition.

4.2.2 Assumption Database

Service assumptions made during the process of service composition are represented as a set of ground formulas and are stored in a database $\mathcal{M}$. We define an assumption database $\mathcal{M}$ to be a 4-tuple $\langle AID, WSID, MA, F \rangle$, where

- AID is the set of all unique identifiers of each primitive assumption (See Section 3.5) in the scope of a particular service composition.
- $WSID$ is the set of all identifiers of Web Services which are produced by the service selection function (See Section 4.1.1) and intend to participate in the Web Service composition.
- MA is the set of all primitive assumptions.
- F is the flag to indicate the assumption status.

In this work, we intentionally use the assumption database $\mathcal{M}$ to store various assumptions made during the process of service composition, instead of writing the assumptions in the state. As we have defined in Section 4.1.4, a service composition planning problem $\mathcal{P}$ is the problem of finding a path or constructing a process of actions given the planning domain initial and goal states. Let $\mathcal{G}$ represent the goal of the service composition, S_g represent the goal state for the service composition, such that S_g is any state such that $S_g \models \mathcal{G}$. The knowledge of service assumption represents the guesses that cover different alternatives, for example: "the weather is sunny" is one possible weather condition among others. Intuitively, we cannot use the assumptions to directly conclude that the goal has been achieved, thus it is necessary to separate state of the knowledge and service assumptions.

Description Logic $\mathcal{L}$ [14] has been adopted as knowledge representation language to describe how service composition problem can be solved. The Semantic Web Service

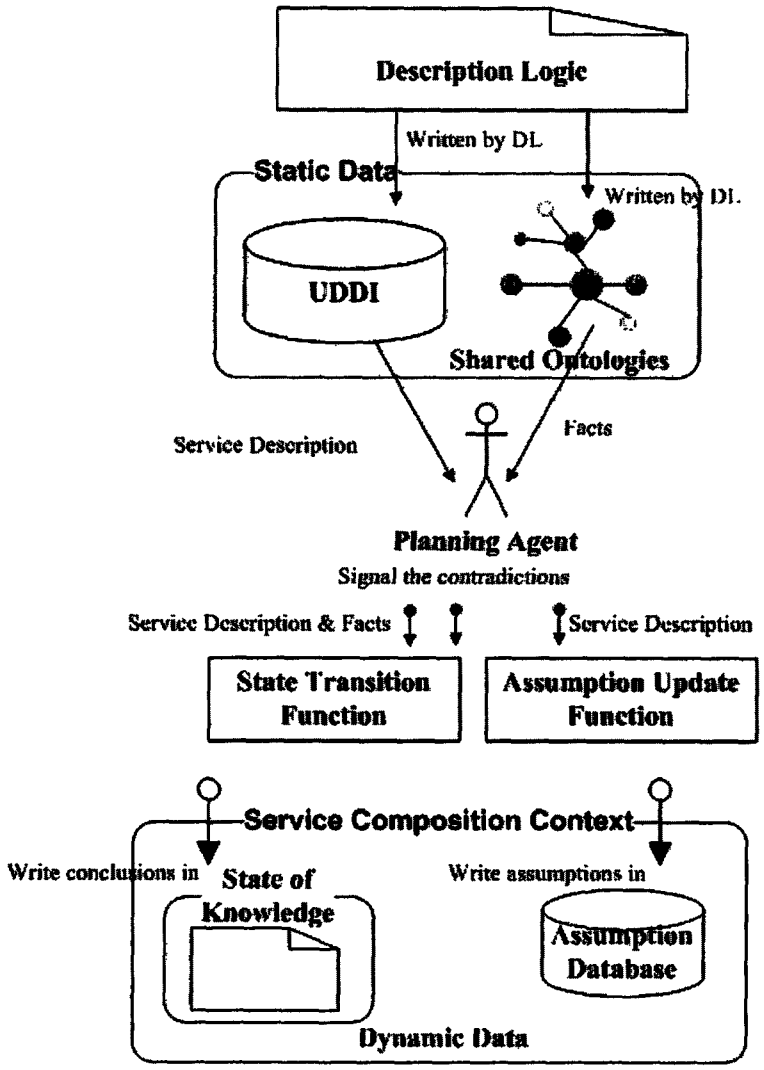

Figure 4.3: Static Data Vs. Dynamic Data

Description, is specified using Description Logic $\mathcal{L}$ with the OWL [23] syntax, thus it is in a logical form that can be used by an inference tool and agent technology to enable automation of services on the Semantic Web.

To guide the planning process toward service composition solutions, a planning agent uses different search strategies (depth-first, bread-first, and so on) for the service matching and selection. In addition, it may also be the case that some well defined ontologies will be used to share information and to facilitate the necessary inference. After the matched service is selected, the state transition function and the assumption

update function are called. Based on the description of the selected service, the state transition function will apply the data to the current state of knowledge S_i, and the assumption update function will update to the assumption database $\mathcal{M}$. However, during this process, if conflicting information is detected, the planning agent must be able to signal these conflicts.

The service descriptions are held in UDDI which contains static data, representing knowledge about the problem solving that is unchanged during the process of service composition. On the other hand, dynamic data generated during the process are maintained in either the state of knowledge or the assumption database, reflecting the shifting of different service composition contexts (See Fig 4.3).

To further clarify different types of knowledge in this framework, we will describe the data used in the service composition planning.

- **Premises** are used to define data that is certainly true. Premises are independent of other data and they are not inferred from other facts. Premises include propositions such as:
  - The opposite of right is left.
  - The speed of light in vacuum is 300,000 km/h.
  - Smoking causes emphysema.
- **Time-Based Facts** are used to define data that is verified to be true at a particular time. For example: *Population of Australia is: 20,264,082* is valid in the statistics years 2006. However, with the *growth rate: 0.9%*, in 2007 this data will not be true anymore. Typically, time-based facts are either produced by the sensing operations or applying the new effect to the current state.
- **Service Assumptions** which means that most instances of a concept have some property. Assumptions are believed in the lack of information to the contrary,

unless the contrary is proved (See Section 3.3). What the service assumptions aim to describe is the environment of service composition.

- **Derived Facts**, which can be inferred using premises, time-based facts, service assumptions, and other derived facts. For example, if the customer has a valid credit card, and he will not use the car for the dune exploration, then the car will be rented to that customer. The proposition "the customer has a valid credit card", represents the precondition and "the customer will not use the car for the dune exploration" represents the assumption. As the service effect, "the car will be rented to that customer" represents a derived fact. In general, every derived fact depends on some other data, and the derived fact is valid if and only if all data it depends on are valid.

- **Justification** are the dependencies that arise between data, i.e. justification describes how derived fact is inferred. Justification represents a relation between a derived fact and its antecedents. In the example above, a justification for the derived fact "the car will be rented to that customer" records that this fact is inferred using "the customer has a valid credit card" and "the customer will not use the car for the dune exploration". For the purpose of handling incomplete knowledge, what we are really concerned is the dependency relation between service effect and service assumption.

State of knowledge is represented by large number of facts and relations, which contains derived fact, time-based facts and premises holds in all states. On the other hand, assumption database describes the environment of service composition, including various service assumptions and corresponding justifications made during the process of service composition (See Fig 4.4). Service assumptions are useful to reason with incomplete information, while justifications are useful to handle the contradictions caused by the underlying service assumptions.

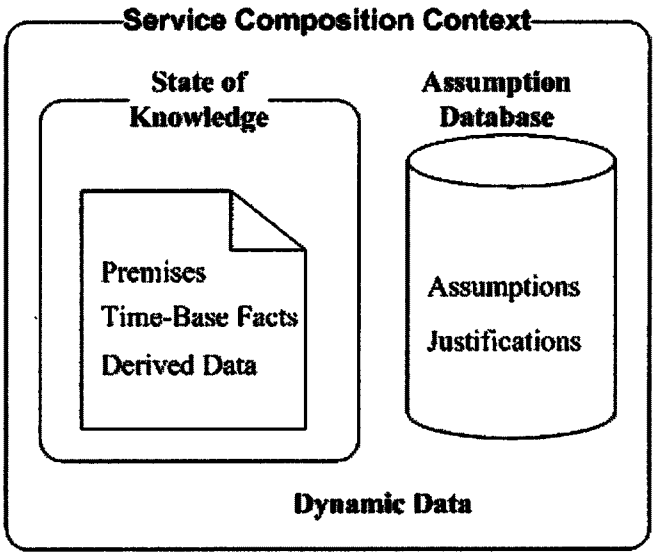

Figure 4.4: Dynamic Data in Service Composition Context

An assumption database $\mathcal{M}$ for a particular service composition problem is defined to be a 4-tuple $\langle AID, WSID, MA, F \rangle$. For any service $ws_i = \langle p_i, e_i, a_i \rangle$, based on the properties of an assumption database $\mathcal{M}$, we can maintain the following information:

- ws_i is identified by $wsid_i$ and $wsid_i \in WSID$.
- the assumption a_i is a set of primitive assumptions. Each of the primitive assumptions is represented by a ground formula, i.e.

$$a_i = \{a_i^1, \ldots, a_i^n\} \text{ and } a_i \subseteq MA.$$

- any primitive assumption a_i^j has its corresponding unique identifier aid_j and $aid_j \in AID$.

The dependencies that arise between a service effect e_i and a service assumption a_i during the process of service composition, are recorded by the combination of AID and $WSID$. The effect e_i, as derived fact, only if the service assumption a_i, on which e_i depends on, is a set of consistent descriptions about the underlying service composition environment, i.e. a contradiction cannot be inferred from the corresponding set of

service assumptions, then the application of e_i to the current state of knowledge is legal. When a contradiction is detected during the process, justifications are used to find all affected service nodes which underlie the contradiction.

Besides recording the dependencies between a service effect e_i and a service assumption a_i, the operation of the assumption database $\mathcal{M}$ has to keep the status of various assumptions up to date. Before we explain the operation for updating the status of service assumptions, we will introduce the concept of outdated assumptions.

There are two cases, in which we refer to a service assumption a_i as an outdated assumption. The first case is very simple. As defined in Section 3.6, based on the relation between e_i and a_i, service assumptions have been classified as transient assumptions and persistent assumptions. If a contradiction can be inferred from the union of a service assumption a_i and its associated effect e_i, i.e, $a_i \cup e_i \models \bot$, then a_i is classified as a transient assumption. For any transient assumption a_i, to avoid nonsensical service descriptions (due to the problem of self-defeating), after its associated effect e_i is applied to the current state of knowledge, a_i is outdated, i.e. we only need to check for consistency beforehand for a transient assumption. For the definitions of transient assumptions and persistent assumption, please refer to the page 51.

In the second case, the effect e_i of service ws_i is a set of sentences, such that $e_i = \{e_i^1, \ldots, e_i^n\}$. We define φ as the negation of the service effect e_i, if $\{\neg e_i^1, \ldots, \neg e_i^n\} \subseteq \varphi$. Here, we use $\neg e_i \subseteq \varphi$ to denote that φ is the negation of the service e_i. If $\neg e_i \subseteq \varphi$ and φ is the logical consequence of a current state of knowledge, we refer to the assumption a_i which is associated with service ws_i as an outdated assumption. In other words, if the negation of all sentences in e_i is entailed by some states S_j, where e_i is an effect of service ws_i and $j > i$, then we refer to the assumption a_i associated with e_i as an outdated assumption. Formally, the assumption is outdated, if $\forall x \in e_i, \exists j > i$ such that $\neg x \in Cn(S_j)$, where $Cn(S_j)$ denotes logical closure of S_j. A simple example of an outdated assumption is: a book borrowing service assumes that a borrower is in same city as the library. When the borrowed book is returned, we say this assumption is

outdated. Outdated assumptions are not allowed to be involved in reasoning process for the consistency of service composition.

To conduct default reasoning about the current state of knowledge for service composition problem, it is necessary to describe and record various assumptions generated during the process of service composition planning. In this framework, we intentionally maintain an assumption database $\mathcal{M}$ which holds the information about these assumptions and their corresponding justifications. Now, we are prepared to introduce the operation for updating the assumption status. The property F of an assumption database $\mathcal{M}$ is a flag which indicates the assumption status, which has three values $\{active, inactive, deleted\}$.

1. *active*: after a service is chosen by the service selection function, and if its service assumption is not classified as a transient assumption, when this service assumption is added to assumption database $\mathcal{M}$, then this assumption's status is initially set to be *active*.

2. *inactive*:

   - when the assumption is outdated. Note that there are two cases in which we refer to the service assumption as being outdated. First, if the service assumption is classified as a transient assumption, after its associated service effect is applied to the current state of knowledge. Second, if the negation of a certain service's effect can be entailed by the logical consequence of the current state of knowledge.

   - when a contradiction is detected during the process, and the contradiction is caused by a soft assumption, then the choice is left to the client. The client could choose to ignore the detected contradiction, then the status of this inconsistent assumption will be set to *inactive*. If any hard assumption is involved in the detected contradiction, it is considered to be a conflict of the Web Service composition anyway. In this case, the client does not have

the control over it, i.e. the contradiction cannot be ignored and it has to be eliminated by revising the current state of knowledge for the given service composition problem.

3. *deleted*: when a contradiction is detected, a retraction may be performed by the planning agent. Simultaneously, the dependency network of service effects and service assumptions maintained by the assumption database are searched using the justification information to find out which assumptions underlie the contradiction. The status of these service assumptions will be set to *deleted*.

Solving the problem of service composition can be considered as searching for a path from an initial to a goal state and the path must not contain inconsistent problem states. During this process, to solve the problem of inconsistency caused by the interactions of multiple independent parities, our solution is to apply a set of rules that describe and perform all permitted state transitions from one problem state to another. Obviously, the rules need to work with various assumptions made during the process. However, only the assumptions whose status is set to be *active* are valid ones to participate in this rule-driven reasoning process. For this purpose, suppose $\mathcal{M}$ is an assumption database for a particular service composition problem, then we use $\Pi(\mathcal{M})$ to denote the set of all active assumptions maintained by $\mathcal{M}$. Let a_i be a sentence maintained under the property A of the assumption database $\mathcal{M}$ (which will be written as $\mathcal{M}.A$) and f_i represent the status of a_i, then we can have

$$\forall a_i \in \mathcal{M}.A, \text{ if } f_i = \textit{active}, \text{ then } a_i \in \Pi(\mathcal{M}) \text{ and } \Pi(\mathcal{M}) \subseteq \mathcal{M}.A$$

4.2.3 State Transition with Rules

To guarantee that the execution of a service composition has the anticipated effects, during the process of service composition, we apply rules that describe and perform all permitted state transitions from one problem state to another. In the proposed system, the state transition $\mathcal{TR}$ is 2-tuple $\langle \mathcal{T}, \mathcal{R} \rangle$, where

1. $\mathcal{T}$ is a transition function $\mathcal{T} : S \times WS \mapsto S$;

2. $\mathcal{R}$ is a sequence of structured rule conditions guide the derivation of new state and govern the state transition function;

3. The atomic service $ws \in WS$ is said to be executable in $s \in S$ if $\mathcal{T}(s, ws) \neq \emptyset$ with the rule conditions R;

In the process of state transition, there are three types of knowledge about the current world. Let SEN_i denote a set of sentences used to change the state S_i. This set of sentences can be partitioned into three categories, namely, state invariant, state expansion and state update. The set of sentences is defined as:

$$SEN_i = \{Inv_i \mid Exp_i \mid Upd_i\}$$

where:

1. State invariant Inv_i denotes a set of sentences which can be entailed by the knowledge in the previous state, defined as:

$$S_{i-1} \models Inv_i$$

2. State expansion Exp_i denotes a set of sentences which cannot be entailed by the knowledge in the previous state and its negation also cannot be entailed by the knowledge in the previous state, defined as:

$$S_{i-1} \not\models Exp_i$$

and

$$S_{i-1} \not\models \neg Exp_i$$

3. State update Upd_i denotes a set of sentences whose negation can be entailed by the knowledge in the previous state, defined as:

$$S_{i-1} \models \neg Upd_i$$

In the last chapter, we have explained the extended semantics of service functional description, which includes precondition, effect and service assumption. Now, we are prepared to define how a Web Service is simulated as an action which changes the state of knowledge. Let ws_i be any Web Service, WS be the set of all Web Services, E be the set of all service effects, P be the set of all service preconditions, we define the following extraction functions:

1. **Effect Extraction Function** $f_e : WS \rightarrow E$ which takes an arbitrary atomic service ws_i as an input, and extracts the effect e_i of ws_i as its output. e_i is a set of primitive effects of ws_i and every primitive effect is a partition with the state invariant, state expansion and state update i.e.

$$f_e(ws_i) = e_i \text{ and } e_i = \{eInv_i \mid eExp_i \mid eUpd_i\}$$

in which $eInv_i, eExp_i, eUpd_i$ denote state invariant, state expansion and state update respectively.

2. **Precondition Extraction Function** $f_p : WS \rightarrow P$ which takes an arbitrary atomic service ws_i as an input, and extracts the precondition p_i of ws_i as its output. As we explained in the Section 4.2.1, the state of knowledge contains incomplete but correct information about the world state, thus the precondition evaluation either depends on the current state of knowledge or is based on sensing operation which adds new knowledge to the current state. In addition, we have assumed that the information gathered cannot be changed by external services in the planning process. Thus the knowledge generated from the sensing operation for the purpose of precondition evaluation can only expand the current state of knowledge. Here, precondition p_i of the service ws_i is defined as a set of sentences and each sentence is a partition with the state invariant and state expansion, i.e.

$$f_p(ws_i) = p_i \text{ and } p_i = \{pInv_i \mid pExp_i\}$$

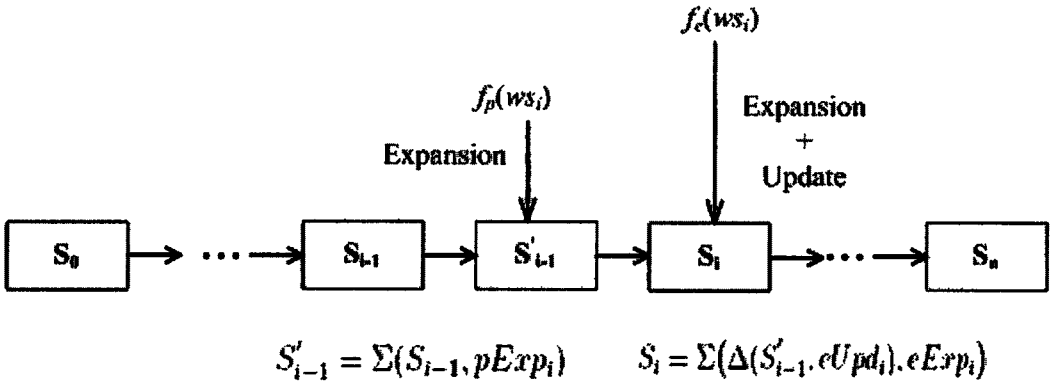

Figure 4.5: Generic State Transition Operators

Following the definitions above, we can define the generic state transition operators (See Fig 4.5) as:

1. $S'_{i-1} = \Sigma(S_{i-1}, pExp_i)$
2. $S_i = \Sigma(\Delta(S'_{i-1}, eUpd_i), eExp_i)$

which means the state transition from S_{i-1} to S_i is completed by means of performing sensing operations for precondition evaluation, then applying the service effect. In step one, the knowledge $pExp_i$ generated from the sensing operations is used to expand previous knowledge of the state S_{i-1}. The operator Σ takes the S_{i-1} and $pExp_i$ as its input, expands knowledge of the S_{i-1} and produces the intermediate state S'_{i-1}. S_i is reached at step two, in which the operator Δ takes the S'_{i-1} and $eUpd_i$ as its input and performs an update to knowledge of the S'_{i-1}. Finally, applying the effect may also lead to knowledge expansion.

Having defined the state transition function, we are prepared to define a sequence of rule conditions which guide the derivation of a new state and govern the state transition function. Compared to the traditional software development, a dynamic service composition is an automated process with less human intervention. Usually, it does not have a predefined boundary, based on which the problems of uncertainty and incompleteness of information could be tackled. Unpredictable service executions

and a dynamically changing context complicate dynamic service composition in many ways. In Chapter 3, we have extended the current Semantic Web Service Description by introducing service assumptions. Together with the proposed service assumptions, a sequence of rule conditions are defined to reason about a changing environment. For any service composition problem ρ, let ws_i represent a Web Service which is produced by the service selection function (See Section 4.1.1) and $ws_i = \langle p_i, e_i, a_i \rangle$, where

- p_i is set of sentences representing the precondition of ws_i, i.e. $p_i = \{p_i^1, \ldots, p_i^n\}$.
- e_i is set of sentences representing the effect of ws_i, i.e. $e_i = \{e_i^1, \ldots, e_i^n\}$.
- a_i is set of sentences representing the assumption of ws_i, i.e. $a_i = \{a_i^1, \ldots, a_i^n\}$.

In addition, $\Pi(\mathcal{M})$ denotes the set of all active assumptions in $\mathcal{M}$ (See Section 4.2.2), where $\mathcal{M}$ is an assumption database used to maintain all service assumptions for the service composition problem ρ. The state transition function takes previous state of knowledge S_{i-1} and Web Service ws_i as the input and produces the new state S_i. To get the legal state transition, inspired by default logics [63, 67], our conflict checking contains three transition conditions:

1. **Precondition Satisfaction** (Cond-A): means that only when a precondition holds, and then the service is a valid candidate service to participate service composition. Formally, if $S_{i-1} \models p_i$, then we define ws_i as a precondition satisfied service. Note that S_{i-1} here also contains the knowledge acquired by the sensing operation for the purpose of precondition evaluation.

2. **Consistency of State and Assumptions**: which means that after the effect e_i of Web Service w_i is applied to the current state, the new state of knowledge S_i must be consistent with the set of all active assumptions $\Pi(\mathcal{M})$ maintained in $\mathcal{M}$. Formally, $S_i \cup \Pi(\mathcal{M}) \not\models \bot$. Normally, e_i is the conclusion of a precondition satisfied service ws_i, but e_i may need to be retracted in face of new evidence. Note that here we intentionally make the design decision that the joint consistency of

service assumptions is required. Thus checking of consistency between the state and the assumptions has two steps:

- **Joint Consistency of Assumptions** (Cond-B): which means the conjunction of all active service assumptions must be consistent. Formally, $\Pi(\mathcal{M}) \not\models \bot$
- **Consistency between State and Assumptions** (Cond-C): which means that in addition to the conjunction of all active service assumptions being consistent, it is also required that the new state of knowledge should be consistent with this set of service assumptions. Formally $S_i \cup \Pi(\mathcal{M}) \not\models \bot$

A state transition $t = \langle S_{i-1}, ws_i, S_i \rangle$ is called legal, if ws_i is a precondition satisfied service with respect to S_{i-1}, the conjunctions of the set of all current active service assumptions is consistent and there is no contradiction which can be inferred from the set of active assumptions with respect to the new state S_i. However, the building of consistent value-added services on a heterogeneous environment is not a trivial task, we have to take in the consideration that

- the current set of service assumptions must be continually updated over time to incorporate new knowledge during this process.
- the conclusions which have been drawn must sometimes be revoked.
- corrections must be soothly accommodated to its corresponding assumptions.

In next section, we will prepare to illustrate the process of constructing a service composition plan and explain how these proposed state transition conditions should be used during the reasoning process.

4.3 Reasoning with Service Assumptions

Service composition planning can be viewed as a process of resolving conflicts and gradually refining a partially specified plan, until it is transformed into a complete plan that satisfies the goal. Service composition planning is similar to the classical planning in that each state of knowledge is represented by a conjunction of ground formulas and each Web Service is related to a transition between those states. However, unlike classical AI planning techniques, in this proposed framework, the planner is the rule based system which allows making tentative conclusions and revising them in face of additional information. In other words, the planner is endowed with the ability to reason about and adapt to a changing environment. As the result of the applying the state transition rules, the generated plan represents an applicable or consistent solution to the service composition problem even with insufficient information during the process. For any state S_{i-1}, Web Service ws_i is not applicable to the state until certain minimal criteria are met. ws_i is specified in terms of the precondition p_i, effect e_i and assumption a_i, where p_i must be satisfied for it to be the precondition satisfied service (Cond-A), the effect may be concluded, however the joint consistency of assumptions (Cond-B) and consistency of new state of knowledge and various service assumptions (Cond-C) are required.

A state in our framework is not a complete view of the world. Usually, an agent is forced to perform sensing operations which aim at finding out the information which could satisfy the precondition p_i. Like "1" shown in Fig 4.6 at page 81, the sensing operation may lead to knowledge expansion of the state S_{i-1}. When the sensing operations complete, if p_i is satisfied, we can conclude that ws_i may be applicable to the current state S_{i-1} (Cond-A). Due to the knowledge expansion to the state S_{i-1}, before the transition to state S_i, we get an intermediate state S'_{i-1}. This intermediate state holds the current state of knowledge after the agent's sensing operation, which is shown as the operation step "2". Following the sensing operations, beforehand checking will

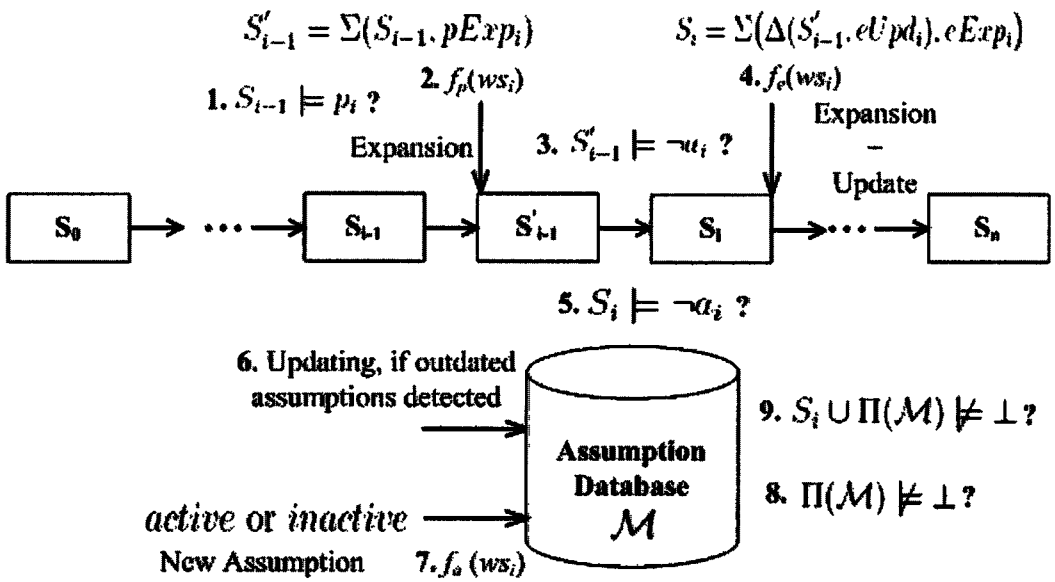

Figure 4.6: Reasoning with Service Assumptions

be performed by means of issuing the query $S'_{i-1} \models \neg a_i$. If $\neg a_i$ is entailed by the knowledge of state S'_{i-1}, which means that the current state of knowledge contradicts the service assumption of new service ws_i. On the other hand, if $\neg a_i$ cannot be entailed by the knowledge of state S'_{i-1}, we say the beforehand checking is successful. This step is shown as operation step "3". If the beforehand checking is successful, effect e_i is applied to the current state to simulate an action. As we mentioned before, the effect e_i may expand and update the knowledge of the current state, which is shown as the operation step "4". This process can be presented as generic state transition operation as we defined in page 77.

After the effect e_i is applied to the current state of knowledge, for this new state of knowledge S_i, it is the time to perform the afterward checking, which could distinguish the type of service assumption a_i. As defined in Section 3.6, based on the relation between e_i and a_i, the service assumptions has been classified as transient assumption and persistent assumption. Because the beforehand checking must have been successful before reaching to this step, i.e. $S'_{i-1} \not\models \neg a_i$, after applying the e_i to the current state

of knowledge, if the $\neg a_i$ is entailed by the new state S_i, i.e, $S_i \models \neg a_i$, which indicates that for a Web Service $ws_i = \{p_i, e_i, a_i\}$, $a_i \cup e_i \models \bot$. In this case, the a_i is classified as the transient assumption, otherwise, the a_i will be classified as a persistent assumption. This step is shown as operation step "5".

One of the main features in this proposed framework is the ability to describe various service assumptions and support default reasoning with these assumptions. The service assumptions generated from the service composition planning are represented as a set of ground formulas stored in the assumption database $\mathcal{M}$. After expanding and updating the knowledge of the current state, the planner needs to carefully perform checking to see whether any outdated assumption is in $\mathcal{M}$, i.e. check whether the negation of any previous applied service effect can be entailed by the logical consequence of the current state of knowledge. Because the outdated assumptions are not allowed to participate in the default reasoning, the status of all outdated assumptions will be set to *inactive*, which is shown as operation step "6". After updating the assumption database $\mathcal{M}$, the service assumption a_i is added to the assumption database $\mathcal{M}$. Based on the operation of step "3" and "5", we have completed the assumption type identification checking. If the service assumption belongs to the type of persistent assumption, then, initially, the status of this new service assumption a_i is set to be *active*, while if this new service assumption is a transient assumption, its status will be set to *inactive*, which is shown as the operation step "7".

Service assumptions are made about things that may not specifically be known during the process of service composition. Thus what the service assumptions represent is the environment of an underlying service composition. Clearly, the combination of the environment and the current knowledge state uniquely identify a service composition context. A particular service composition environment is described by the set of all active assumptions $\Pi(\mathcal{M})$ maintained in the assumption database $\mathcal{M}$. Logically, this environment refers to a conjunction of service assumptions. To achieve consistent service composition, we intentionally make the design decision that the service

composition environment is required to be consistent, which means that there is no contradiction can be inferred from $\Pi(\mathcal{M})$. A consistent service composition environment is enforced by Cond-B which is shown as the operation step "8". Note that the checking of joint consistency of assumptions is performed after both the effect e_i is applied to the current state of knowledge and the updating of all the detected outdated assumptions in the assumption database $\mathcal{M}$ is complete. If a contradiction appears, it means that the service composition environment is no longer consistent and corrections to these assumptions must be made in the face of this contradicting information. The conclusion of applying the e_i of w_i to the state of knowledge must be revoked.

On the other hand, if the the service composition environment is described by a consistent set of service assumptions, the next reasoning task is to check the consistency between the new state of knowledge S_i and the set of all active service assumptions $\Pi(\mathcal{M})$, which is is enforced by Cond-C and shown as operation step "9". This task is completed by means of checking whether the negation of any active assumptions can be entailed by the current state of knowledge. The negation of a service assumption plays the role of being a defeater, which prevents the effects associated with this assumption being applied to the state. Similarly, if the contradicting information is detected at this step, it means that the previous conclusions are not appropriate in the face of this additional information and the old conclusions must discarded in order to incorporate new knowledge and adapt to a changing environment. Up to now, the process of state transition from S_{i-1} to S_i is completed. We have illustrated that how the new state of knowledge is reached in the presence of possibly incomplete or conflicting information.

Notice that, although the are treated as the same during the process of service composition planning, there is a fundamental difference between the ways of handling the conflicts caused by hard assumptions and soft assumptions respectively. Typically, when the conflict is detected, and the conflict is caused by a soft assumption, the choice will be left to client. The client could choose to ignore the detected conflict. However, if the detected conflict is produced by a hard assumption, it is considered to be a conflict

of the Web Service composition anyway, and the client does not have control over it.

Chapter 5

Scenario

This chapter will present different scenarios in which the application of service assumption to Semantic Web Service Description might prove to be a useful to achieve the consistent service composition. In addition, these scenarios could ease the intelligibility of the reasoning process with the proposed service assumptions during service composition planning.

Our example uses the often presented travel agency service package. A typical use case could involve arranging a trip comprising a hotel booking, a car rental and a sightseeing service. To simplify this use case, we assume this composite service is executed in sequential manner (i.e. hotel booking service, then car rental service, finally sightseeing service).

5.1 Scenario One

The first scenario demonstrates that, when service composition is built in an open and distributed environment, service assumptions can be used to represent the information which may not be specifically known in the context of service composition, and how the violation of a service assumption can be detected. In this example, the service conflict arises when composing a car rental service and a sightseeing service together as part of a travel agency service package. Assume that when requesting this composite travel agency service, the user specifies his preferred car model, for example, a city car.

Obviously, this car will be used for sightseeing which is also generated as part of this composite service. If the functionality matches the user's requirement, then a car rental service is invoked. In the real world, it is most likely that the car rental service providers have some service policy about usage of rental cars. However, when the car rental service is invoked, we don't have any information about what kinds of sightseeing plan might be generated from the execution of the service, in other words, we don't know how the rented car will be used. The point here is that different sightseeing plans may be associated with different roads, and it may not be allowable for a rented car to drive on certain roads. For example, a desert dune exploration plan is dynamically generated from the service and a city car is used for the desert dune exploration. Clearly, this is not an acceptable situation for either the car rental company or the customer.

Thus, to ensure integrity of service composition, there should be a mechanism to deal with incompleteness of information during dynamic service composition. Our solution to this problem is to use service assumptions. In this example, to prohibit the illegal usage of the rental car, the car rental service could make the assumption that "city cars do not drive on dune, beach, unsealed road...". If the contrary evidence appears (e.g. a dune exploration) from the succeeding service executions, then we can conclude that there is a violation to the car usage policy. In other words, if we can get additional information which is explicitly contradictory to the service assumptions in the context of the service composition, then the potential service conflicts are detected.

```
(:Web Service  Car Rental
 :parameters (?cust - Customer
              ?car  - Car Model
              ?ccn  - Credit Card No
:precondition (and (?cust hasCreditCard ?vc)
                   (isValid ?ccn))
:effect (?cust rented ?car)
:assumption (driveCarInProperWay ?car))
```

Code Segment 5.1.1: Car Rental Service

Suppose the travel package service composition is requested by a customer. Also at this stage, we suppose that the hotel booking service has been successfully applied as part of requested service composition. The next task for the service composition planning is to find a car rental service. After searching in the service registration UDDI, the Car Rental Service (See Code Segment 5.1.1) is located and generated as the output from service selection function. The selected Car Rental Service which is intended to participate in the requested service composition, is described as follows:

Precondition: $hasCreditCard(?cust, ?ccn) \wedge isValid(?ccn)$

Effect: $rented(?cust, ?car)$

Service Assumption: $driveCarInProperWay(?car)$

Satisfaction of the Service Precondition: To participate in the service composition, the selected service must be a precondition satisfied service, here Cond-A is applied. Note that a state in our framework is not a complete view of the world. Usually, for the purpose of precondition evaluation, it is necessary to perform sensing operations which aim to find out the information which could satisfy the precondition. In the example above, to be a valid candidate service, the precondition associated with this Car Rental Service must be satisfied, that is:

$$hasCreditCard(?cust, ?ccn) \wedge isValid(?ccn)$$

Applying the Effects: If the car rental service is a precondition satisfied service, normally the effects associated with this service can be applied to the current state, i.e. $rented(?cust, ?car)$. However, these effects may be defeated in the face of new information because Cond-B or Cond-C may apply.

Making assumptions: Certainly, the rented car should be used in proper way. For instance, if the rented car is a city car, common sense dictates that this car should not be used for a mountain or desert dune exploration. To deal with exceptions which may result from uncertainty, the car rental service provider makes an assumption here that the car is not to be used for certain road conditions. In the example above, the service assumption is represented by a SWRL Rule, and $driveCarInProperWay(?car)$ is the

consequent of SWRL Rule. Correspondingly, the antecedent of the rule consists of two primitive assumptions, i.e. $\neg driveOn(?car, dune)$ and $\neg driveOn(?car, unsealedRoad)$. Thus the assumption is equivalent to a conjunctive query, and it would be read as:

$$\neg driveOn(?car, dune) \wedge \neg driveOn(?car, unsealedRoad) \Rightarrow driveCarInProperWay(?car)$$

Note that "?car" here is a variable, while "dune" and "unsealedRoad" represent OWL individuals. After Car Rental Service is invoked, we get the the variable bindings for "?car". Because "dune" and "unsealedRoad" represent OWL individuals which are facts about individual identity, thus they are independent on service requester's selection, i.e. no substitution or binding for "dune" and "unsealedRoad".

After applying the Car Rental Service, the current state of knowledge holds the following the sentences: $hasCreditCard(cust, ccn) \wedge isValid(ccn) \wedge rented(cust, car)$. In addition, the assumption database needs to maintain the service assumption "the rented car drives in the property way". In this case, the two primitive service assumptions are added to the assumption database and their status is set to *active*. Note that at this time, "?car" has been bound to an actual value, thus it is written as "car" instead of "?car", i.e.

$$\neg driveOn(car, dune) \wedge \neg driveOn(car, unsealedRoad)$$

```
(:Web Service Sight Seeing
 :parameters (?cust - Customer
              ?ssp - Singht Seeing Plan
:precondition (and (?cust hasCar  ?car)
                   ...))
:effect (and (?cust isAssigned ?ssp)
             (?car driveOn RdCon))
:assumption (NULL))
```

Code Segment 5.1.2: SightSeeing Service

To clarify the usage of the assumption and the reasoning with assumptions in our framework, here we give a simplified example of a sightseeing plan service (See Code Segment 5.1.2), which is supposed to integrate with the car rental service together

as a travel agency package. After successfully requesting the Car Rental Service, the customer chooses a rented car to register to the sightseeing plan service. The desert dune exploration as a sightseeing plan is dynamically generated. As a result, the rented car will be used for this dune exploration.

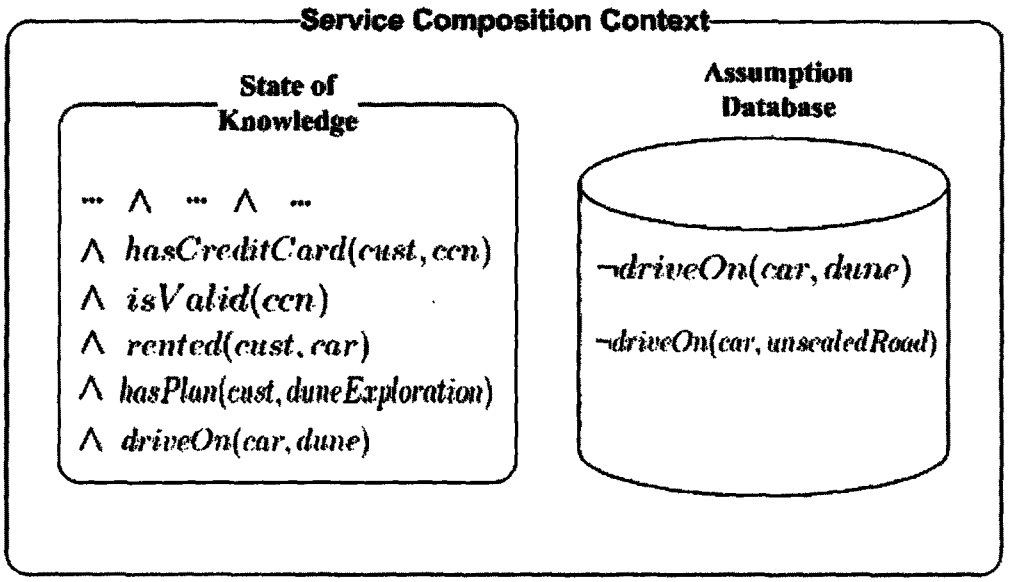

Figure 5.1: Current State of Knowledge and Assumption Database

The service assumptions generated during the process of service composition are stored in the assumption database, which allows reasoning with uncertainty and dealing with exceptions. In our example, the service assumption is used by the service provider to enforce the policy about a rental car usage, which contains $\neg driveOn(car, dune) \wedge \neg driveOn(car, unsealedRoad)$. After applying the Sightseeing Service, the current state of knowledge S_i and the assumption database is showed in Fig 5.1. This new state of knowledge contains the sentence of $driveOn(car, dune)$. Clearly, it contradicts the *active* Car Rental service assumption $\neg driveOn(car, dune)$, that is $S_i \cup \Pi(\mathcal{M}) \models \bot$. Thus the consistency condition Cond-C failed. In this case, the violation against the car usage is detected, the car will not be rented to the customer for his desert dune exploration, therefore the policy about the legal rental car usage is enforced. However, one might hold the view that the policy about the legal rental car usage can be enforced

by a precondition and why bother to use the service assumption. To answer this question, we need to distinguish the differences of the semantics between precondition and service assumption. The precondition denotes the condition that must be satisfied for the atomic service to execute, while for the service assumption, we only need to establish that it is consistent with what is known i.e. nothing is known that contradicts this service assumption. Intuitively, in our Car Rental Service example, we cannot expect the information about the rental car usage to be available in **ALL** service compositions in which the car rental service participates. Obviously, precondition is not a good way to represent this information. Service composition is a dynamic behavior, before the service composition is complete, the service requester may not know everything about the underlying service composition to be, so do the planning agents. To describe such uncertain information and deal with the exceptions, using service assumption is ideal way to make service description more accurate and precise.

5.2 Scenario Two

The second scenario presented still use travel agency service package as the example, which demonstrates that service conflict may be caused by service assumptions made by different independent parties. In this example, the service conflict arises when composing a hotel booking service, a car rental service and a sightseeing service together as the part of travel agency service package. Assume that when requesting this composite travel agency service, the first service located by the service planning agent is the hotel booking service, without any explicit information about the service requester's preference, one of luxury hotel booking services is located (See Code Piece 5.2.1).

```
(:Web Service Hotel-Booking Service
 :parameters (?cust - Customer
              ?dt - Required Date
              ?rm - Room
:precondition (and (?rm isAvailable ?d)
                   ...))
```

```
:effect (and (?cust booked ?rm))
:assumption (?cust isNotBugetTraveller))
```

Code Segment 5.2.1: Hotel Booking Service

The selected hotel booking service which is intended to participate in the requested service composition, is described as follows:

Precondition: $isAvailable(rm, dt)$

Effect: $booked(cust, rm)$

Service Assumption: $\neg bugetTraveller(cust)$

This description of the hotel booking service simply says that if on the required date, there is still more room available for booking, and nothing is known about the customer being budget traveller, then customer books the room in hotel. Typically a budget traveller won't consider a luxury hotel as an ideal accommodation. To avoid such an situation, the hotel booking service simply makes this assumption. However, this assumption is set as a soft assumption, whose purpose is to provide warning information instead of restricting the service usage. After the effect of hotel booking service is applied, its assumption $\neg bugetTraveller(cust)$ is maintained in the assumption database and its status is set to *active*.

```
(:Web Service Car Rental
 :parameters (?cust - Customer
              ?car  - Car
              ?ccn  - Credit Card No
:precondition (and (?cust hasCreditCard ?vc)
                   (isValid ?ccn))
:effect (?cust rented ?car)
:assumption (budgeTraveller ?cust)
                        (driveCarInProperWay ?car))
```

Code Segment 5.2.2: Car Rental Service

The next task for the planning agent will be finding the Car Rental Service. Because there is no explicit preference given by service requester, an extreme example could be that a rental service is located (See Code Piece 5.2.2), but unfortunately this

service only provides the out-of-fashion car and normally most of its customers are budget travellers. The service description of this Car Rental Service as follows:

Precondition: $hasCreditCard(cust, ccn) \wedge isValid(ccn)$

Effect: $rented(cust, car)$

Service Assumption: $budgeTraveller(cust)$

This description of the Car Rental Service simply says that if the customer has valid credit card, and the given credit card is a valid one, and nothing is known about the customer not being a budget traveler, then customer rented the car. Out of the consideration that, typically, a luxury traveler won't consider renting an out-of-fashion car, the Car Rental Service simply makes the assumption $budgetTraveller(cust)$. Like the service assumption made by the hotel booking service, this service assumption is designed as a soft assumption, whose purpose is to provide the warning information instead of restricting the service usage. In other words, if a luxury traveler intentionally chooses a budget car, nothing will prevent him from renting this car. After the effect of the Car Rental Service is applied, the service assumption is added into assumption database and its status is set to *active*. What service assumptions describe is the environment of an underlying service composition, in other words, the environment of the service composition refers to a conjunction of service assumptions. An environment is consistent if a contradiction cannot be inferred from the corresponding set of service assumptions (Cond-B). In this example, the condition of joint consistency of assumptions is violated (See Fig 5.2), that is $\Pi(M) \models \perp$.

Notice that although being treated as the same during the process of service composition planning, there is a fundamental difference between the way of handling the conflicts caused by hardAssumptions and SoftAsumptions. In this example the detected conflict information is caused by soft assumptions made by Hotel Booking Service and Car Rental Service respectively, the choice will be left to the service requester. The service requester could choose to ignore the detected service conflicts.

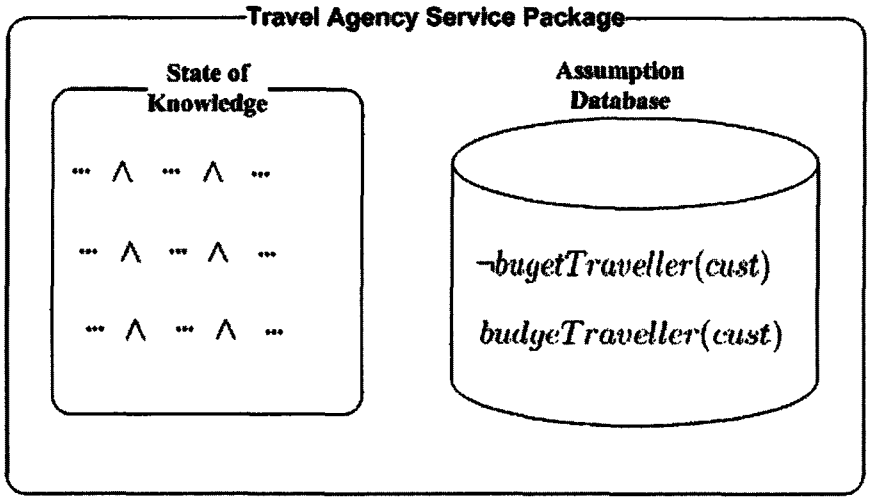

Figure 5.2: Violation of Joint Consistency of Service Assumptions

5.3 Summary

In many situations, the service composition has only incomplete information at the planning or run time, perhaps because some information is unavailable, or because it has to respond first before obtaining all the relevant information. Service description is described by OWL-S which is based on Description Logic indeed has the capacity to help users and agents search, discover, invoke, compose and monitor Web Services. However, there are occasions where plausible conjectures need to be "filled in" to overcome the incompleteness of information during the process of service composition. In such situations, service assumption can help a service composition planner make more accurate decisions. For example, in the field of health informatics, the online health symptom checking service has to make some assumptions about the most likely causes of the symptoms observed before reaching the final result. Obviously, it would be impossible to have all information about this patient at the beginning. For some emergency situations, it is also inappropriate to wait for the results of possibly extensive and time-consuming tests before starting treatment.

One of differences between service precondition and service assumption is that precondition can be viewed as the conditions regarding the eligibility of a service to be

used, while the service assumption can be viewed as the conditions regarding the way in which a service is used. For some services, if service providers care about the usage of their services, service assumption is an ideal way to put such constraints upon the service usage. In our example of Car Rental Service, obviously, it is inappropriate to use the precondition to prevent the rented car from the illegal usage, because precondition is logical conditions that only need to be satisfied before service is invoked, in other words, after satisfaction of this precondition and service is invoked, the precondition no longer has the control over the usage of rented car. On the other hand, because of the character of being persistent, after the invocation of Car Rental Service, service assumption can continually monitor the usage of the rented car in the life cycle of underlying service composition.

Chapter 6

Conclusion and Future Work

In this chapter, firstly, we will review the proposed extensions to the current Semantic Web Service Description. Then we will discuss how these extensions are used in the underlying framework which meets the needs of service composition in a changing environment. Based on this analysis, we will outline how this work can progress further.

This work has focused on design and development of a framework which allows services to have the ability to adapt to a changing service composition environment. Normally, in an open environment, the action must be taken in the presence of incomplete knowledge or uncertainty. In terms of service composition, incompleteness arises when anything regarding a changing environment is not explicitly represented in the service composition context. Dealing with incomplete information in the service composition context is a practical, complicated and challenging problem in the field of Web Service composition.

OWL-S [4] is a formal language which aims to provide precise and rich declarative specification of a wide variety of properties about Web Services in order to support automation of a broad spectrum of activities across the Web Service life cycle, such as discovery, selection, composition, negotiation and contracting, invocation and monitoring of progress. In the current version of OWL-S, the vast majority of efforts and techniques focus on the modeling and specification of the Web Service alone. Certainly, the declarative specifications of the prerequisites, consequences of application of individual services, and data flow interactions need to be defined precisely and related to

each other. In current version OWL-S, these properties of services have been defined by means of $\langle I, O, P, E \rangle$. However, in the absence of service assumptions which are essential for a service to have the capability of adapting to a changing environment, the service composition specification offered by OWL-S is incomplete or inaccurate. Thus, in parallel, the assumptions made about incomplete knowledge also need to be explicitly represented and documented as an indispensable part of service composition specification.

In this work, we have extended OWL-S to a richer service description representation schema by introducing service assumptions. The general goal of adding service assumptions as one property of a Web Service is to allow making plausible inferences in the process of service composition and ensure consistent service composition, which might be seen as:

- accurately describing the service composition environment, in which most instances of a concept generally have some property, but not always.
- presenting the hypothetical guesses about incompleteness and uncertainty.
- some combination of both.

The goal of dealing with incomplete information in the service composition context is certainly a challenging task. In our proposed framework, together with the proposed service assumption, we developed a sequence of rule conditions for reasoning with various assumptions during the process of service composition planning. We also illustrated how knowledge based planning could reason about incomplete knowledge in the service composition context and construct a service composition plan. During the planning process, we showed that only when a precondition holds, then the service is a valid candidate service to participate service composition. Specially, by adopting service assumptions, the framework supports default reasoning in the presence of incomplete knowledge. The service assumptions are made about the things that may not

specifically know during the process of service composition, thus what the service assumptions represent is the environment of a underlying service composition. Logically, this environment refers to a conjunction of service assumptions. To achieve the consistent service composition, we intentionally make the design decision that the service composition environment is required to be consistent. Finally, consistency between the state of knowledge and the set of all active service assumptions is required. This consistency checking task is completed by the means of checking whether the negation of any active assumptions can be entailed by the current state of knowledge. The negation of a service assumption plays the role of being a defeater, which prevents the effects associated with this assumption being applied to the state. Briefly, this proposed framework allows us to make tentative conclusions based on the available information, and to detect potential conflicts in service composition when further suitable information about the problem is available. This proposed work also leaves many opportunities for future improvements, which include:

1. Moving beyond sequential service composition, how the underlying service assumptions can be used to deal with incomplete knowledge and uncertainty in distributed parallel processing.

2. To facilitate the further automation of web service composition, it is desirable to have priorities among various service assumptions. These priorities indicate the different levels of the preference that the service requester would have on any given service composition. With these explicitly specified priorities, the planning agents may be able to re-compile their knowledge in response to conflicting information or partial failures.

Bibliography

[1] Resource Description Framework (RDF): Concepts and Abstract Syntax, Graham Klyne and Jeremy J. Carroll, Editors, W3C Recommendation, 10 February 2004, http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/ . Latest version available at http://www.w3.org/TR/rdf-concepts/.

[2] RDF Vocabulary Description Language 1.0: RDF Schema, Dan Brickley and R. V. Guha, Editors, W3C Recommendation, 10 February 2004, Latest version available at
http://www.w3.org/TR/rdf-schema/ .

[3] The DAML Coalition. http://www.daml.org.

[4] "OWL-S White Paper OWL Services Coalition". OWL-S: Semantic markup for Web services, 2005. http://www.daml.org/services/owl-s/1.1/overview

[5] UDDI. The UDDI technical white paper, 2000. http://www.uddi.org/.

[6] WSDL, http:// www.w3.org/TR/wsdl, 15 March 2001

[7] http://www.wsmo.org/

[8] http://www.w3.org/Submission/WSML

[9] Web Services Architecture http://www.w3.org/TR/ws-arch/

[10] V. Andrea and M. Aiello. Services and objects: Open issues In G. Piccinelli and S. Weerawarana, editors, European workshop on OO and Web Service, pages 23–29, 2003. IBM Research Report. IBM. Computer Science, (RA 220).

[11] Antoniou G. The role of nonmonotonic representations in requirements engineering. Int J Software Eng Knowledge Eng 1998;8(3):385-3399.

[12] G. Antoniou (1999). A tutorial on default logics. ACM Computing Surveys, 31(4):337-359.

[13] Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., und Weerawarana, S. 2003. Business Process Execution Language for Web Services, Version 1.1. Specification. BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems.

[14] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. 2003. The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press 2003.

[15] Benjamin N. Grosof, Ian Horrocks. 2003. Description Logic Programs: Combining Logic Programs with Description Logic" ACM 1581136803/03/0005.

[16] Tim Berners-Lee, James A. Hendler, and Ora Lassila. The Semantic Web. Scientific American, 284(5):34–43, May 2001.

[17] Bertoli, P., Cimatti, A., Dal Lago, U., and Pistore, M. 2003. Extending PDDL to nondeterminism, limited sensing and iterative conditional plans. In ICAPS Workshop on PDDL, Informal Proceedings, pages 15-24.

[18] Chapman, D. 1987. Planning for conjunctive goals. Artificial Intelligence 32(3): 333-377.

[19] Cooke DE, Luqi . Logic programming and software maintenance. Ann Math Artif Intell 1997;21:221-229.

[20] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S.Weerawarana. Unraveling the web services web: An introduction to SOAP, WSDL, and UDDI. IEEE Internet Computing, 6(2):86-C93, 1 2002.

[21] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Service v1.1. Published online by BEA, IBM and Microsoft at http://www.ibm.com/developerworks/library/wsbpel, May 2003.

[22] Dardenne,A., A. van Lamsweerde and S. Fickas. 1993. Goal-Directed Requirements Acquisition, Science of Computer Programming, 20, pp. 3-50

[23] Dean, M. and Schreiber G. 2004. OWL Web Ontology Language. Reference W3C Recommendation, http://www.w3.org/tr/owl-ref/.

[24] Deborah L. McGuinness; Ontologies Come of Age. Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential. MIT Press. 2003

[25] Doyle, J. 1979. A truth maintenance system. Artificial Intelligence, 12: 231-272.

[26] Easterbrook.S. M. 1991. Handling Conflict Between Domain Descriptions With Computer Supported Negotiation. Knowledge Acquisition: An International Journal, Vol 3, No 4, pp 255-289.

[27] Easterbrook.S. M., Beck, E., Goodlet,J., L. Plowman, M. Sharples, and C. C. Wood (1993) A Survey of Empirical Studies of Conflict. In S. M. Easterbrook (ed) CSCW: Cooperation or Conflict? London: Springer-Verlag, pp1-68.

[28] Erol, K.; Hendler, J.; and Nau, D. 1994. HTN Planning: Complexity and Expressivity. In Proceedings of the Twelfth National Conference on Artificial Intelligence, 1123-1128. Menlo Park, Calif.: American Association for Artificial Intelligence.

[29] Etzioni, O., Hanks, S., Weld, D., Draper, D., Lesh, N., and Williamson, M. (1992). An approach to planning with incomplete information. In Nebel, Bernhard; Rich,

Charles; Swartout, W., editor, Proceedings of the 3rd International Conference on Principles of Knowledge Represen tation and Reasoning, pages 115-125, Cambridge, MA, USA. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.

[30] Fikes, R., and Nilsson, N. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. Journal of Artificial Intelligence 2(3-4): 189-208

[31] Ginsberg, M. 1993. AI and nonmonotonic reasoning. In Handbook of Logic in Artificial Intelligence and Logic Programming: Vol. 3: Nonmonotonic Reasoning and Uncertain Reasoning. Edited by D. Gabbay, C. Hogger, and J.A. Robinson. Oxford University Press, Oxford, pp. 1-33.

[32] Golden, K., Etzioni, O. & Weld, D. 1996. Planning with Execution and Incomplete Information, UW Technical Report TR96-01-09, February 1996.

[33] Golden, K. & Weld, D. 1996. Representing Sensing Actions: The Middle Ground Revisited, Proc. 5th Int. Conf. on Principles of Knowledge Representation and Reasoning

[34] Howard Foster, Sebastián Uchitel, Jeff Magee, Jeff Kramer. Compatibility Verification for Web Service Choreography. Proceedings of the IEEE International Conference on Web Services (ICWS04), San Diego, California, USA. IEEE Computer Society pp. 738-741, 2004

[35] R. V. Guha. Contexts: A Formalization and Some Applications. PhD thesis, Stanford University, 1991.

[36] D. McDermott and AIPS'98 IPC Committee. PDDL–the planning domain definition language. Technical report, Available at: www.cs.yale.edu/homes/dvm, 1998.

[37] R.J. Hall, Open Modeling in Multi-stakeholder Distributed Systems: Model-based Requirements Engineering for the 21st Century, in 2002 Workshop on the State

of the Art in Automated Software Engineering, U.C.Irvine, Institute for Software Research

[38] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A Semantic Web Rule Language - Combining OWL and RuleML. W3C Member Submission, http://www.w3.org/Submission/SWRL/, May 2004.

[39] M. Jackson, Software Requirements & Specifications: A Lexicon of Practice, Principles, and Prejudices, Addison- Wesley, Wokingham, England, 1995.

[40] Knoblock, C. 1995. Planning, Executing, Sensing, and Replanning for Information Gathering. In Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, 1686-1693. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

[41] Kreger, H. Web Services Conceptual Architecture (WSCA 1.0). http://www-4.ibm.com/software/solutions/webservices/, 2001

[42] U. Kuter, E. Sirin, D. Nau, B. Parsia, and J. Hendler. Information gathering during planning for web service composition. In Proceedings of 3rd International Semantic Web Conference (ISWC2004), Hiroshima, Japan, November 2003.

[43] A. van Lamsweerde, R. Darimont, E. Letier "Managing Conflicts in Goal-Driven Requirements Engineering" IEEE Transactions on Software Engineering, Special Issue on Managing Inconsistency in Software Development, November 1998

[44] LUKASZEWICZ, W. 1988. Considerations on default logic. Comput. Intell. 4, 1, 1-16

[45] Luqi , Cooke DE. How to combine nonmonotonic logic and rapid prototyping to help maintain software. Int J Software Eng Knowledge Eng 1995;5(1):89-118

[46] S. McIlraith and T. C. Son. Adapting Golog for composition of Semantic Web service Proceedings of the 8th International Conference on Knowledge Representation and Reasoning (KR2002), Toulouse, France, April 2002.

[47] R. F. Neches, R.; Finin, T.; Gruber, T.; Patil, R.; Senator, T.; Swartout, W.R., "Enabling Technology for Knowledge Sharing., AI Magazine36-56, 1991.

[48] Nuseibeh, B. and Easterbrook, S. Requirements Engineering: A Roadmap, International Conference on Software Engineering,Proceedings of the Conference on The Future of Software Engineering, Limerick, Ireland, Pages: 35 - 46, 2000

[49] Bashar Nuseibeh, Steve Easterbrook and Alessandra Russo. Leveraging Inconsistency in Software Developoment. IEEE Computer, 33(4):24-29, April 2000.

[50] Bashar Nuseibeh and Steve Easterbrook and Alessandra Russo. Making inconsistency respectable in software development. The Journal of Systems and Software volume 58-2 171–180, 2001

[51] M. Paolucci, Takahiro Kawamura, Terry R. Payne, Katia Sycara. "Importing the Semantic Web in UDDI". In Proceedings of Web Services, E-business and Semantic Web Workshop.

[52] M. Paolucci, T. Kawmura, T. Payne and K. Sycara. Semantic Matching of Web Services Capabilities. In First Int. Semantic Web Conf., 2002

[53] M.P. Papazoglou and J. Yang, "Design Methodology for Web Services and Business Pro- cesses", Procs. of the 3rd VLDB-TES Workshop, August, Hong Kong, Lecture Notes in Computer Science Vol. 2444, Springer, 2002

[54] M. P. Papazoglou and D. Georgakopoulos. Service oriented computing. Commun. ACM, Oct. 2003.

[55] M.P. Papazoglou, Extending the Service Oriented Architecture, Business Integration Journal, February 2005,

[56] Simon Parsons,.; and Anthony Hunter. 1998. A review of uncertainty handling formalisms. Lecture Notes In Computer Science; Vol. 1455. 8 - 37. Springer-Verlag. ISBN:3-540-65312-0

[57] Paulo F. Pires, Mário R. F. Benevides, and Marta Mattos "Building Reliable Web Services Compositions", Net.Object Days - WS-RSD'02, page 551-562, 2002

[58] Pednault, E. . ADL and the state-transition model of action. Journal of Logic and Computation. 1994

[59] Penberthy, J., and Weld, D. 1992. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning, 103-114. San Francisco, Calif.: Morgan Kaufmann.

[60] W. Poon and A. Finkelstein, "Consistency Management for Multiple Perspective Software Development," presented at ACM SIGSOFT 96 Workshop - Viewpoints 96, 1996.

[61] Putnam, L. L., and M. S. Poole (1987) Conflict and Negotiation. In L. W. Porter, Ed., Handbook of Organizational Communication: An Interdisciplinary Perspective, pp. 549-599, Newbury Park: Sage.

[62] Reiter, R. 1978. On reasoning by default. In Proceedings of TINLAP-2, Association for Computatinal Linguistics, University of Illinois, pp. 210-218.

[63] Reiter R. "A logic for default reasoning", Artif Intell 1980; 13:81132.

[64] Robbins, S. P. (1989) Organizational Behavior: Concepts, Controversies and Applications. Englewood Cliffs, NJ: Prentice-Hall.

[65] W.N. Robinson, I Didn't Know My Requirements were Consistent until I Talked to My Analyst, Living With Inconsistency Workshop at The 19th International

Conference on Software Engineering, IEEE Computer Society Press, Boston, USA (May 17-24 1997).

[66] Russel, S. and Norvig, P. (2002). Artificial Intelligence: A Modern Approach. Prentice-Hall Inc.

[67] SCHAUB, T. 1992. On constrained default theories. In Proceedings of the 10th European Conference on Artificial Intelligence (ECAI92, Vienna, Austria, Aug. 3-7), B. Neumann, Ed. John Wiley and Sons, Inc., New York,NY, 304-308.

[68] M. Smithson. Ignorance and Uncertainty. Emerging Paradigms. Springer Verlag. New York. NY. 1989

[69] T. Sollazzo and S. Handschuh and S. Staab and M. Frank. Semantic Web Service Architecture - Evolving Web Service Standards toward the Semantic Web. Proceedings of the 15th International FLAIRS Conference, Pensacola, Florida, May 16–18, 2002. AAAI Press

[70] Spanoudakis G., Zisman A.: Inconsistency Management in Software Engineering: Survey and Open Research Issues, Handbook of Software Engineering and Knowledge Engineering, (eds) Chang S. K., World Scientific Publishing Co., 2001.

[71] Evren Sirin, Bijan Parsia, Dan Wu, James Hendler, and Dana Nau. HTN planning for web service composition using SHOP2. Journal of Web Semantics, 1(4):377-396, 2004

[72] Biplav Srivastava and Jana Koehler. Web Service Composition - Current Solutions and Open Problems in Proceedings of ICAPS03, 2003

[73] Stefan Tang. MATCHING OF WEB SERVICE SPECIFICATIONS. USING DAML-S DESCRIPTIONS. Thesis Diplomarbeit. March 18th, 2004

[74] TANIMOTO,S.L.The Elements of Artificial Intelligence. Computer Science Press, Rockville, MD, 1987

[75] Weld, D. S.; Anderson, C. R.; and Smith, D. E. 1998. Extending GRAPHPLAN to Handle Uncertainty and Sensing Actions. In Proceedings of the Fifteenth National Conference on Artificial Intelligence, 897-904. Menlo Park, Calif.: American Association for Artificial Intelligence.

[76] Wiktor Marek and Mirek Truszczyński. Nonmonotonic Logics; Context-Dependent Reasoning. Springer, Berlin, 1st edition, 1993.

[77] Jian Yang.; Mike P. Papazoglou. Web Component: A Substrate for Web Service Reuse and Composition. In Proceedings of the 14th International Conference on Advanced Information Systems Engineering, 21 - 36, 2002. Springer-Verlag, London, UK ISBN:3-540-43738-X

[78] J. Yang,. ; M.P. Papazoglou,: Service Components for Managing the Life-Cycle of Service Compositions. Will appear in Information Systems, June, 2003.

[79] Zowghi D, Ghose A, Peppas P. A framework for reasoning about requirements evolution. In: Proc 4th Pacific Rim Int Conf AI. Lecture Notes in Artificial Intelligence 1114. New York: Springer-Verlag; 1996. pp 157-168.