Self-healing Web Service Composition with HTN Planners

by

Ka Sim May Chan

Submitted in partial fulfilment of the requirements for the degree
Magister Scientia (Computer Science)
in the
Faculty of Engineering, Built Environment and Information Technology
University of Pretoria
Pretoria

2008

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Self-healing Web Service Composition with HTN Planners

by

Ka Sim May Chan

Abstract

Web services have become a prominent paradigm for building of both inter- and intra-enterprise business processes. These processes are composed from existing Web services based on defined requirements. Standards and techniques have been developed to aid in the dynamic composition of services. However, these approaches are limited when it comes to the handling of unexpected events.

This dissertation presents the results of experiments that investigated numerous problems related to Web service composition processes. Based on the investigation, a fault taxonomy was formulated. Faults were grouped into three broad categories, each representing a distinct problem stage. The investigation into faults gave rise to the issue of fault recovery and continued process execution. A list of requirements for self-healing Web service composition was identified, while a new self-healing cycle was exploited based on the MAPE cycle (**M**onitor, **A**nalyzer, **P**lanner, **E**xecutive). The proposed self-healing composition cycle consists of three modules: Plan Generation Module, Plan Execution Module and Failure Analysis Module. The plan execution module, consisting of the execution and run-time monitoring phases, and the failure analysis module, consisting of the analysis and sensemaking phases, were found to be vital to self-healing Web service composition. Self-healing Web service composition and the goal of self-healing were achieved through the use of Hierarchical Task Network (HTN) planning systems.

Keywords: Web service composition, fault taxonomy, self-healing, HTN, planning, verification, execution monitor, analyser, sensemaking, replanning

Supervisor: Prof. J.M. Bishop
Co-supervisor: Prof. T.J. Grant
Department of Computer Science
Degree: Magister Scientia (Computer Science)

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Acknowledgements

My sincere thanks to:

- my parents and my brother for their unconditional support;
- Professor Judith Bishop and Professor Tim Grant for their professional and thorough supervision;
- my friends for giving me courage to continue;
- Ezra for his coding assistance;
- Luciano Baresi and Sam Guniea in Politecnico di Milano, Italy for the discussions they provided;
- the members of the Polelo research group, especially Johnny, Johan and Pierre-Henri, for their valuable feedback and suggestions;
- the members of ICSA, Espresso and CIRG research groups and TechTeam for the conversations they supplied;
- South African National Research Foundation and DST South African-Italy Research Grant for their financial assistance.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Background

Web services are increasingly used as the key building blocks for creating inter- and intra-enterprise business processes. Such business processes are created through the composition of several existing Web services in a logical order that satisfies certain requirements. Several low-level process modelling and execution standards have been proposed to compose Web services. One of the most prominent standards for composition is Business Processes Execution Language (BPEL) [38]. Such a standard allows developers to implement composite services using simple mono building blocks. The positions setup that uses these standards mostly requires manual implementation by the developers. Given the scale of the Internet and the constant changing environment, manual composition is restrictive, inflexible, time consuming and often error prone. For example, if one of the participating services becomes unavailable due to server downtime, the composition plan will not be able to select an alternative service to replace it. It may be forced to abort the composition or the composition may not behave as expected. Therefore, automated composition is desirable to cope with dynamisms and reduce errors.

Recent research [33], [31], [49], [45], [46], [18], has identified that a service composition problem can be viewed as a planning problem, and that such planning problems can be solved by using Artificial Intelligent (AI)

planning systems. In particular, Hierarchical Task Network (HTN) planning systems such as SHOP2 have been successfully applied in Web service composition [49]. Although AI planning systems have the ability to generate plans automatically, the correctness of such plans during execution cannot be guaranteed without monitoring and control. Therefore, self-healing plays a crucial role in ensuring correctness throughout plan execution. Self-healing is defined as the capability of a system to autonomously detect failures and recover from them. It is also one of the attributes of Automatic Computing defined by IBM [26]. The intention of a self-healing system is to improve resilience by avoiding disorders through discovery, diagnosis and reaction to unexpected working conditions. The control cycle (MAPE cycle) is the key to the self-healing problem. It consists of four integrated phases and must be able to

1. ***Monitor*** its own behaviour in order to detect service delivery failure;
2. ***Analyse*** these failures in order to diagnose the faults causing them;
3. ***Plan*** proper fault remediation strategies; and
4. ***Execute*** these plans in order to restore the normal behaviour of the system.

The objective of self-healing service composition is to minimize the number or duration of outages in order to maintain the availability of the composite service.

Even though the MAPE cycle is used by various applications in self-healing systems, it has its limitations when it comes to Web service composition. One main drawback is that when a service fails to deliver, it will perform analysis and try to find another plan. However, replanning might not be necessary if the service was delayed due to a slow network. In this case, a re-execution of the composition might solve the problem. Given this, an alternative self-healing cycle is needed for Web service composition.

The preceding paragraphs have provided a brief overview of the problems related to Web service composition, with details to follow. The aim of this

work is to investigate the problems related to Web service composition and to determine a set of requirements which Web service composition should fulfil in order to be able to self-heal. Lastly, an evaluation is made of the HTN planning systems with respect to the proposed requirements to show their applicability in self-healing Web service composition.

1.2 Terminology

A number of terms used frequently in this dissertation are defined below.

- A ***plan*** is a composition workflow.
- A ***service*** is a self-contained, self-describing, open component that supports rapid, low-cost composition of distributed applications.
- A ***composition*** is the process of combining existing services.
- A ***workflow*** defines the potential flow of control and data amongst a set of services.
- ***Planning*** is the process of generating a composition workflow.
- A ***planning system***, also known as a ***planner***, is a system that generates the composition workflow. It is also responsible for the execution of the workflow generated. Some examples of planning systems are SIPE-2 [53], O-Plan [17] and JSHOP2 [27].
- An ***execution monitor*** is a mechanism that is used to monitor the execution of the composition workflow. It is responsible for reporting any abnormal behaviour that deviates from the one that is expected.
- ***Replanning*** is the process of workflow modification so as to overcome the problems detected by the execution monitor.
- ***Sensemaking*** is the ability to make sense of an ambiguous situation.

1.3 Problem Statement

We have chosen to explore the opportunity of HTN planning systems in self-healing Web service composition. The investigation of problems in terms of current approaches in Web service composition forms the basis of the author's proposed fault taxonomy. In conjunction with the fault taxonomy, the requirements for a self-healing Web service composition are also needed to ensure that all the necessary actions are taken in reaction to the occurrence of unexpected events.

The first challenge faced is to develop a fault taxonomy that captures all the possible causes and effects of a fault occurring during composition. The taxonomy is used to identify the critical problems and/or most frequently occurring problems during composition. Next an investigation has to be conducted into the exact requirements for self-healing Web service composition. This will be followed by an investigation into the ways in which HTN planning systems can be applied to self-healing Web service composition. Experiments will be conducted as proof-of-concept in answer to the problems posed.

1.4 Research Methodology

To start off, the current state of Web service composition will be explored in a literature survey. Various composition problems will be examined in detail to identify a set of requirements that self-healing composition must fulfil. Once the requirements have been identified, a self-healing composition cycle will be constructed to satisfy the requirements that were determined.

Hierarchical Task Network (HTN) planning systems are to be evaluated according to these self-healing requirements. The purpose of the theoretical evaluation of the HTN planning systems is to show the viability of the self-healing composition cycle, as well as the capability of HTN planning systems in self-healing Web service composition. Lastly, the theoretical evaluations will be substantiated through practical experiments. The experiments are conducted with the aim of gauging how the HTN planning systems withstand

both the normal and abnormal behaviour of a composition process.

1.5 Research Contributions

The contributions made to research by this dissertation can be summarised as follows:

- A novel fault taxonomy for Web service composition (Chapter 3: Section 3.3). The taxonomy is used to classify possible faults that may occur during the composition of services.
- A list of requirements for self-healing Web service composition, and a self-healing composition cycle (Chapter 3: Section 3.4). This cycle is used to aid the evaluation of the applicability of HTN planning in self-healing Web service composition (Chapter 4).
- An online shopping domain system (Chapter 5: Section 5.2). The results of the experiments have shown that HTN planning can indeed be used in self-healing Web service composition.

1.6 Overview

This dissertation consists of six chapters and has the following structure:

Chapter 1 – Introduction: This chapter describes the basic concepts of planning in relation to Web service composition and their related issues.

Chapter 2 - Fundamentals of Composition: A basic background to (semantic) Web services and composition of service is provided in this chapter. Current standards as well as their problems in service composition are discussed. The chapter ends with a brief discussion on the application and suitability of AI planning techniques in Web service composition.

Chapter 3 - Failures and Self-healing: We start this chapter with the classification of possible errors that could occur during the composition process. Following this classification, we examine the role of and the need for self-healing Web service composition. Furthermore, a set of requirements

that is needed to fulfil the self-healing Web service composition process is deduced.

Chapter 4 - Evaluation of HTN Planners: This chapter presents the evaluation of HTN planners in self-healing Web service composition.

Chapter 5 - Experiments: This chapter presents the experimental results showing an application of the HTN planning systems in self-healing Web service composition.

Chapter 6 - Conclusions and Recommendations: This chapter concludes the research conducted. The summary of the dissertation is presented along with recommendations for possible future research.

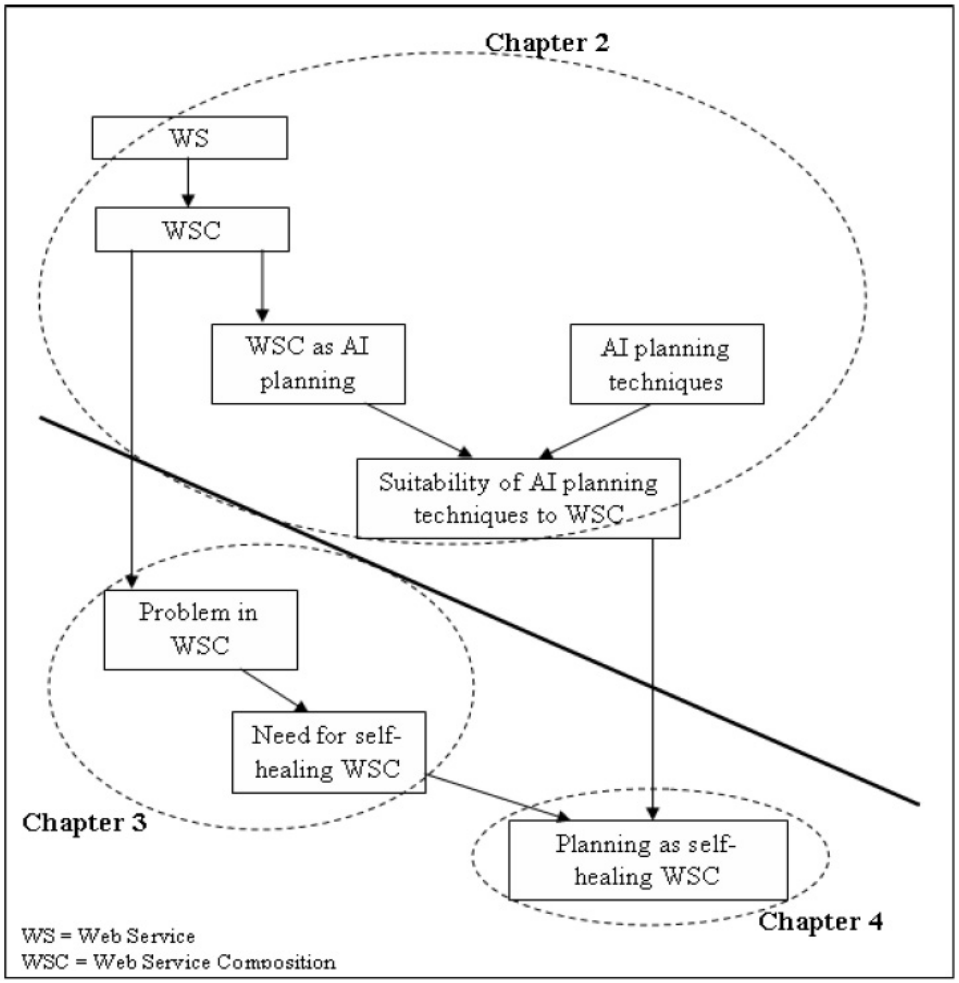

Figure 1.1: Dissertation Outline

Chapter 2

Fundamentals of Composition

2.1 Introduction

This dissertation starts off by explaining why current approaches to service composition are not sufficient to ensure the correctness of the composite service and how planning can assist in setting up the composition plan. The AI planning techniques and their application in Web service composition are explained, with special attention to planning in the dynamic environment, which is important in run-time services composition. Finally, the chapter is concluded with the application of AI planners to dynamic Web services composition.

2.2 Web Services

> *A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols.* - [5]

A Web service is a service that embodies the paradigm of Service-Oriented Computing (SOC) [42], where services are utilised as fundamental elements

for developing applications. To build the service model, SOC relies on Service-Oriented Architecture (SOA) [44], which is an architectural approach aimed at facilitating standard, simple and pluggable interfaces between applications. Applications offered by different providers as services can be discovered, composed, and coordinated in a loosely coupled manner.

Service providers describe their service interface by using the Web Service Description Language (WSDL) [16]. They publish the interface on the online repository such as Universal Description, Discovery, and Integration (UDDI) [10], for discovery. A WSDL file is an XML document that describes a Web service in two fundamental stages, the abstract and concrete stage. In the abstract stage, the description defines what the Web service consists of, that is, the service interface. The interface is comprised of operations and input/output formal parameters. Tags are mostly used for abstract description and their usage is summarised in Table 2.1.

Table 2.1: Tags for abstract WSDL description

Tags	**Definition**
<portType>[1]	The operation performed by the Web service.
<operation>	Web service function.
<message>	Message used by the Web service. Represents collections of input or output parameters.
<type>	The data types of the message parts.
<part>	Incoming or outgoing operation parameter data.

The concrete stage describes the access specification on how the Web service is bound to the set of concrete protocols and its location. Tags are mostly used for concrete description and their usage is summarised in Table 2.2.

[1]In WSDL version 2.0 specification the name of this element has been changed from <portType> to <interface>.

Table 2.2: Tags for concrete WSDL description

Tags	**Definition**
<binding>	Defines the communication protocol used by the Web service.
<service>	Defines collection of endpoints.
<port>[2]	Contains endpoint data, including physical address and protocol information.

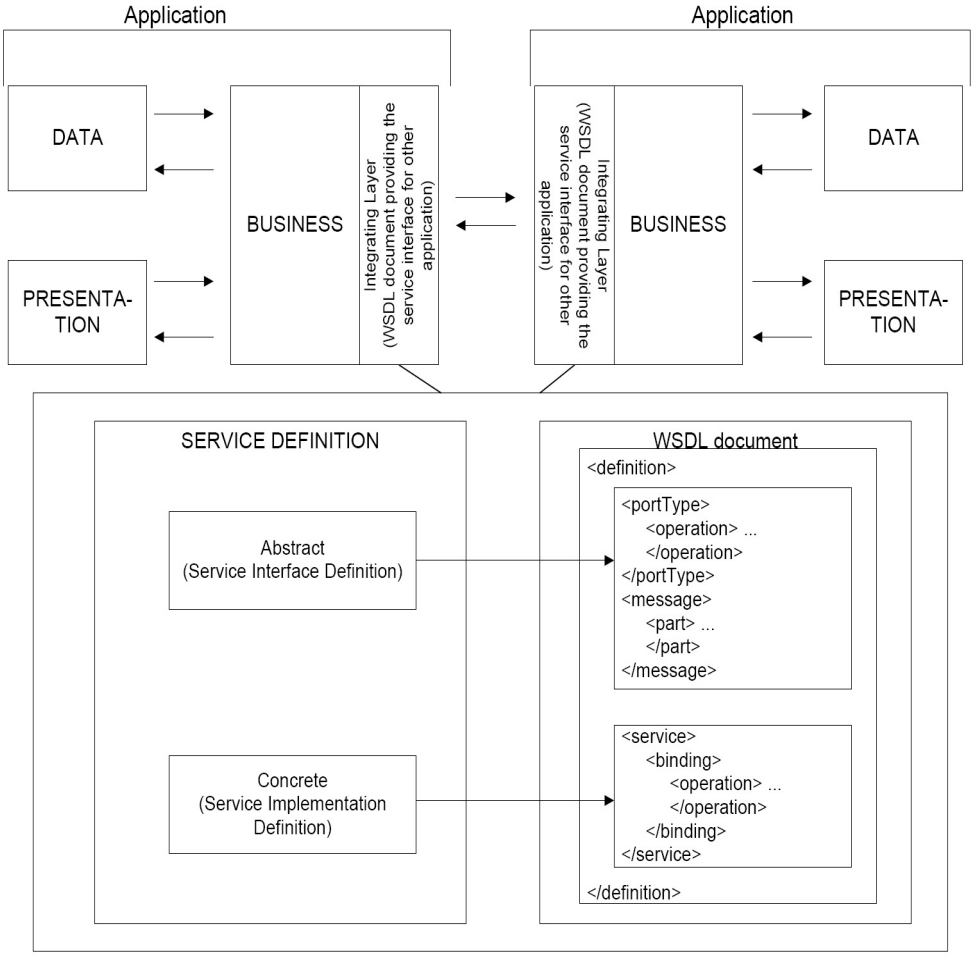

Figure 2.1: WSDL

[2]In WSDL version 2.0 specification the name of this element has been changed from <port> to <endpoint>.

Figure 2.1 depicts how businesses offer an application as a service online. The WSDL document is used to describe the interface of the service that will be made available online.

As has already been mentioned, UDDI is a directory service where businesses can register and search for Web services. The communication protocol used to register and search for Web services is the Simple Object Access Protocol (SOAP) [24]. SOAP messages contain XML data content and the service description of the Web services. The XML data content specifies the structure or grammar of documents that describe Web services and the service description of the Web service is presented as a WSDL file.

In general, Web services can be classified into two categories, namely simple (atomic) and complex (composite) services [34]. The simple services are individual Web services that have the ability to provide some functionality on their own. The complex services are value-added services that combine several existing simple or complex Web services, possibly from different providers, providing the ability to establish more powerful applications.

2.2.1 WSDL Example

Listing 2.1 shows a fragment of a simple credit approver service document specified in WSDL 1.1.

```
<definitions >

  <message name="approvalMessage">
    <part name="accept" type="xsd:string"/>
  </message>

  <portType name="creditApprovalPT">
    <operation name="approve">
      <input message="creditdef:creditInformationMessage"/>
      <output message="tns:approvalMessage"/>
      <fault name="creditProcessFault"
             message="creditdef:creditRequestErrorMessage"/>
    </operation>
  </portType>

  <binding name="SOAPBinding" type="tns:creditApprovalPT">
    <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="approve">
```

```
      <soap:operation soapAction="" style="rpc"/>
      <input>
         <soap:body use="encoded" namespace="urn:creditapprover"
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
       </input>
       <output>
         <soap:body use="encoded" namespace="urn:creditapprover"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
       </output>
      </operation>
   </binding>

   <service name="CreditApprover">
     <documentation>Credit Approver Service</documentation>
     <port name="SOAPPort" binding="tns:SOAPBinding">
       <soap:address location=
                "http://localhost:8080/bpws4j-samples/servlet/rpcrouter"/>
     </port>
   </service>

</definitions>
```

Listing 2.1: Credit Approver Service Description in WSDL 1.1

This document describes a Web service `CreditApprover` that provides a single request-response operation `creditApprovalPT` (line 7). The operation has an input message called `creditdef:creditInformationMessage` (line 9) and an output message called `tns:approvalMessage` (line 10). The input message that the service received is the client's credit information sent by the client to the server. The corresponding output message sent by the server to the client is an approval message stating whether the credit application has been approved or not. Data types of the incoming and outgoing messages are defined under <`message`> element using <`part`> and <`type`>. All these descriptions are part of the service interface definition (abstract description).

The concrete description (service implementation definition) is defined by the <`binding`> element. This element defines the communication protocols and message formats as `name=SOAPBinding` and `type=tns:creditApprovalPT` (line 16) respectively. The `soap:binding` element is defined with two attributes: `style` and `transport`. The `style` attribute indicates whether the operation approved is RPC-oriented or document-oriented. In this example `rpc` is chosen. If the operation is RPC-oriented, it means the messages con-

tain parameters and return values, whereas in the case of a document-oriented operation, the messages will only contain document(s). The `transport` element defines the SOAP protocols to use, while `transport="http://schemas.xmlsoap.org/soap/http"` (line 18) shows the corresponding HTTP binding in the SOAP specification.

2.3 Semantic Web Services

Although a WSDL document describes the interface, endpoint and protocol of the Web services, it does not provide the machine-understandable information of what the service does. In other words, the operations and messages of the service described in WSDL are only understandable to humans or custom-developed applications. The machine cannot distinguish two services that have the same number of input and output messages.

It is clear to us that using WSDL documents alone to describe a service will only allow manual discovery and composition of Web services. Manual composition of Web services is not sufficient for keeping up with the dynamic and fast changing pace of the Internet [20]. Service descriptions specified in WSDL lack the semantic expressivity and cause difficulty in differentiating between the services that appear to be syntactically identical to each other but differ semantically. With manual composition, there is no way to check the semantics of the service and it may cause binding of functionally incompatible services. In order to facilitate automatic Web services discovery and composition, the service should be described in a way that is unambiguous and machine-understandable.

The solution is to describe services in a semantic Web ontology [12]. Semantic Web, an extension of the current Web, is based on ontology, which provides an exact description of Web information and its relationships. By using semantic Web, the information can be processed by computers and the integration of information can be performed when necessary. Web service semantics (WSDL-S) [2] aim to extend WSDL to incorporate semantic annotations to the service description. Another well-known language for semantic service description is Web Ontology Language (OWL) [32], an XML-based

language that was designed to provide a common way to read and process the content of Web information by computers. In particular, OWL ontology consists of a set of classes, instances and properties which describes the concepts of a particular domain.

OWL-based Web service ontology refers to an ontology that provides OWL classes and properties suitable for describing the semantics of Web services.

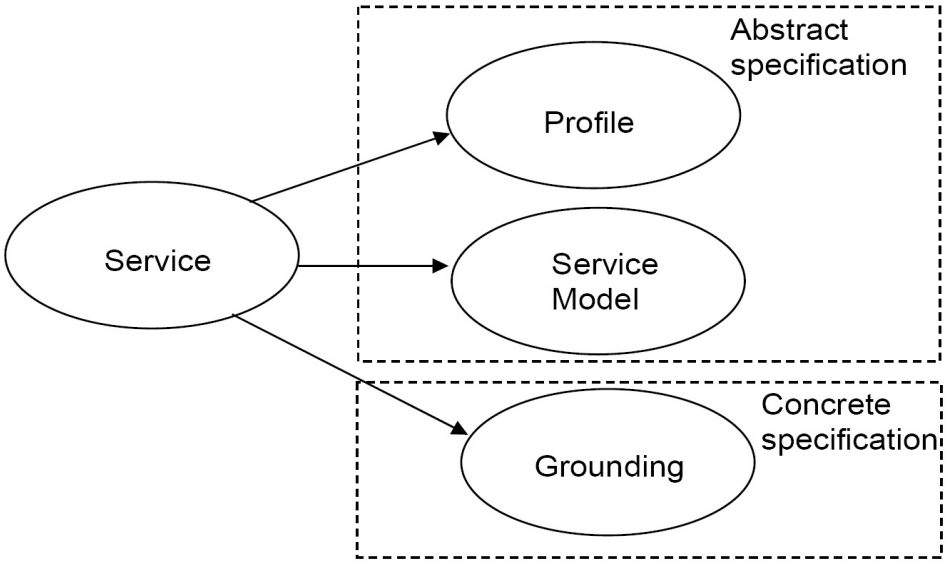

Figure 2.2: Ontology description of service

Figure 2.2 depicts the description of a service class that displays the following properties:

- *Profile*: describes what the service does in terms of its functional and non-functional properties. The information provided by the profile of a service is used to advertise the service capabilities for the purpose of discovery.

- *Service Model*: describes how the service works in terms of the order of service execution. More specifically, it is the invocation of other (atomic or composite) services, the inputs, outputs, precondition and effects of the execution. The service model is mainly concerned with services composition.

- *Grounding*: describes how the service can be accessed. The main concern is the mapping of abstract specifications to a concrete specification, which is the WSDL description.

The mapping from WSDL to OWL-S [41] is done in three phases:

1. Translate the XSD (XML Schema Definition) types in WSDL specification into the corresponding OWL-S ontologies (shown in Table 2.3).

Table 2.3: Mapping between WSDL XSD and OWL-S Ontologies

WSDL XSD	**OWL-S Ontologies**
Primitive XSD (e.g. string and integer)	Input and output of atomic process
Complex XSD	Concepts which specifies the content of input and output

2. Map the WSDL operations to the OWL-S atomic processes. Figure 2.3 depicts the operation translation between WSDL and OWL-S and a summary of the mapping is presented in Table 2.4.

Table 2.4: Mapping between WSDL Operation and OWL-S Atomic Process

WSDL Operation	**OWL-S Atomic Process**
portType	Primitive Process Model
Name of the operation	Name of the atomic process
Input messages of the operation	Inputs of the atomic process
Outputs and faults of the operation	Outputs of the atomic process

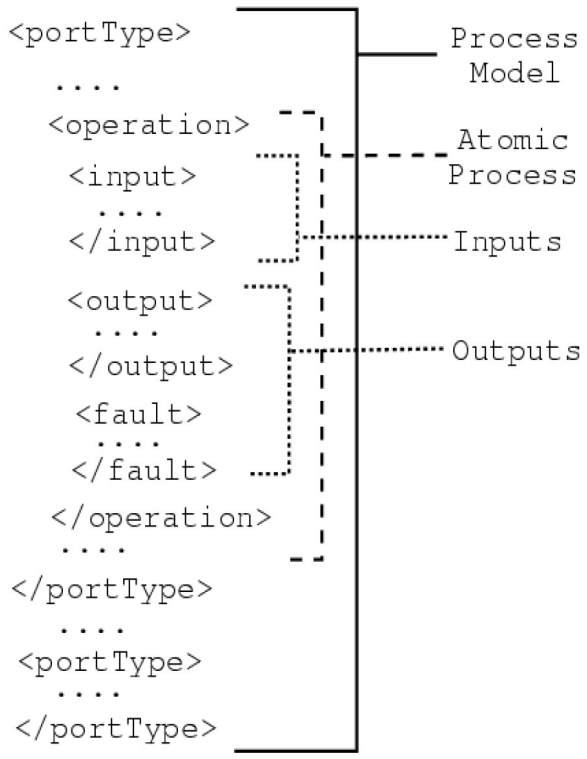

Figure 2.3: Operation Translation [41]

3. Generate service profiles. The generation process is further divided into three stages:

   - The *capabilities* of the Web service.
   - *Provenance information* that describes the entity (i.e. the person-/company that deployed the service).
   - *Non-functional parameters*, such as the quality rating of the service, which describes the services.

   Since WSDL only provides input and output, the above-mentioned information will have to be implemented manually.

2.3.1 OWL-S Example

An equivalent Web Ontology Language for Services (OWL-S) [30] description, converted from WSDL to OWL-S using the WSDL2OWLS tool [55] of the Credit Approver (as was described in Section 2.2.1), is given in Listing 2.2.

The `CreditApproval` service description consists of three parts, namely `CreditApproval-Profile` (line 6), `CreditApproval-Process-model` (line 6), and `CreditApproval-Grouding` (line 7). The detailed description of the service profile is given between lines 11 and 23. The service takes a Credit Card information (line 19) as an input and output the result, stating whether or not the credit application has been approved to the user (line 20). During the credit approval phase, an automic process (that is the `CreditApproval-Profile`) is invoked to return the results to the user. Inside this automic process there are two outcomes, and one of them will be selected based on the decision made. Either the credit is approved (lines 34 - 45) or the credit is not approved (lines 46 - 58). The detailed description of the input to the `CreditApproval` service is defined as a process itself, called `Input` (lines 61 - 67). Similarly, the output to the `CreditApproval` service is defined as an `Ouput` process (lines 69 - 75). Lastly, the grounding of the `CreditApproval` service is given in lines 77 - 79.

```
<owl:Ontology rdf:about="">
    <owl:imports rdf:resource="http://127.0.0.1/ontology/Concepts.owl"/>
</owl:Ontology>

<service:Service rdf:ID="CreditApproval">
    <service:presents rdf:resource="#CreditApproval-Profile"/>
    <service:describedBy rdf:resource="#CreditApproval-Process-Model"/>
    <service:supports rdf:resource="#CreditApproval-Grounding"/>
</service:Service>

<profile:Profile rdf:ID="CreditApproval-Profile">
    <service:isPresentedBy rdf:resource="#CreditApproval-Service"/>
        <profile:serviceName xml:lang="en">
          Credit Approval
    </profile:serviceName>
    <profile:textDescription xml:lang="en">
          Receives credit card info and checks credit approval.
    </profile:textDescription>
    <profile:hasInput rdf:resource="#CreditCard"/>
```

```
    <profile:hasOutput rdf:resource="#CreditCheckResult"/>
    <profile:hasResult rdf:resource="#CreditApproved"/>
    <profile:hasResult rdf:resource="#CreditNotApproved"/>
</profile:Profile>

<process:ProcessModel rdf:ID="CreditApproval-Process-Model">
     <service:describes rdf:resource="#CreditApproval-Service"/>
     <process:hasProcess rdf:resource="#CreditApproval-Process"/>
</process:ProcessModel>

<process:AtomicProcess rdf:ID="CreditApproval-Process">
   <process:hasInput rdf:resource="#CreditCard"/>
   <process:hasOutput rdf:resource="#CreditCheckResult"/>
   <process:hasResult>
      <process:Result rdf:ID="CreditApproved">
          <rdfs:comment>Credit Card Approved</rdfs:comment>
          <process:Effect>
              <expr:SWRL-Expression>
                  <expr:expressionBody rdf:parseType="Literal">
                      <swrl:AtomList rdf:about=
                       "http://www.w3.org/1999/02/22-rdf-syntax-ns#nil" />
                  </expr:expressionBody>
              </expr:SWRL-Expression>
          </process:Effect>
      </process:Result>
   </process:hasResult>
   <process:hasResult>
      <process:Result rdf:ID="CreditNotApproved">
          <rdfs:comment>Credit Not Approved</rdfs:comment>
          <process:Effect>
              <expr:SWRL-Expression>
                  <expr:expressionBody rdf:parseType="Literal">
                      <swrl:AtomList rdf:about=
                       "http://www.w3.org/1999/02/22-rdf-syntax-ns#nil" />
                  </expr:expressionBody>
              </expr:SWRL-Expression>
          </process:Effect>
      </process:Result>
   </process:hasResult>
</process:AtomicProcess>

<process:Input rdf:ID="CreditCard">
    <process:parameterType
     rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
     http://127.0.0.1/ontology/Concepts.owl#CreditCard
    </process:parameterType>
    <rdfs:label>Credit Card Information</rdfs:label>
</process:Input>

<process:Output rdf:ID="CardCheckResult">
```

```
    <process:parameterType
      rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
      http://127.0.0.1/ontology/Concepts.owl#ValidationResult
    </process:parameterType>
    <rdfs:label>Credit Card Validation Check Result</rdfs:label>
</process:Output>

<grounding:WsdlGrounding rdf:ID="CreditApproval-Grounding">
    <service:supportedBy rdf:resource="#CreditApproval-Service"/>
</grounding:WsdlGrounding>

</rdf:RDF>
```

Listing 2.2: Credit Approver Service Description in OWL-S

2.4 Web Services Composition

Service composition refers to the process of creating customised services from existing services. This is done by a process of the dynamic discovery, integration and execution of those services in a deliberate order to satisfy user requirements [14]. Service composition involves interactions between all participating Web services. Figure 2.4 depicts the interaction between different parties, namely the service provider, service broker and service requestor. Service providers[3] export the services they provide so that the service requestors can discover the service that best fits their quality requirements. Service requestors discover services by sending their requirements to the service broker. The service brokers are located between the service providers and service requestors. Their functionality is to collect and advertise available services and to facilitate interaction and the matching process according to the requirements set by the requestor. Once service requests match the requirements, point-to-point interactions will occur between service requestors and service providers. The point-to-point interaction between service providers and services requestors initiates the composition process.

According to Chakraborty and Joshi [13], Web service composition can be classified into two broad categories: the first is proactive and reactive composition; the second is mandatory and optional service composition.

[3]Service providers can also be service requestors and vice versa.

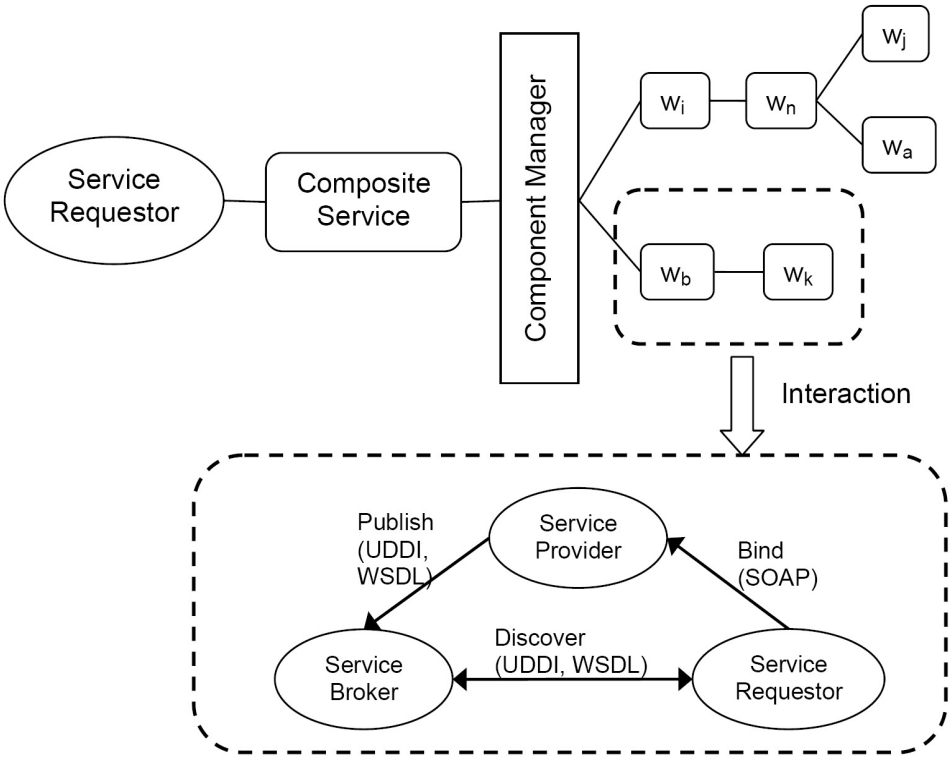

Figure 2.4: Web Service Composition Model

Proactive (static) composition means offline or pre-compiled composition. Services that are composed proactively are usually stable and used at a very high rate over the Internet. *Reactive (dynamic) Composition* means creating a compound service on the fly. Unlike the proactive composition, predefined interaction is not possible and it varies according to the dynamic situation. A component manager can be used to coordinate the interactions between the different subservices and provide the composite service to the service requestors. The component manager should exploit the present state of services and provide certain run-time optimisations based on real-time parameters like bandwidth and the cost of execution of the different subservices.

Mandatory service composition means that all the subservices must participate in the proper execution of composition. These types of services will be dependent on the successful execution of other subservices to produce a satisfactory result. Optional service composition is the opposite of manda-

tory service composition and does not necessarily need the participation of certain subservices for the successful execution of a query.

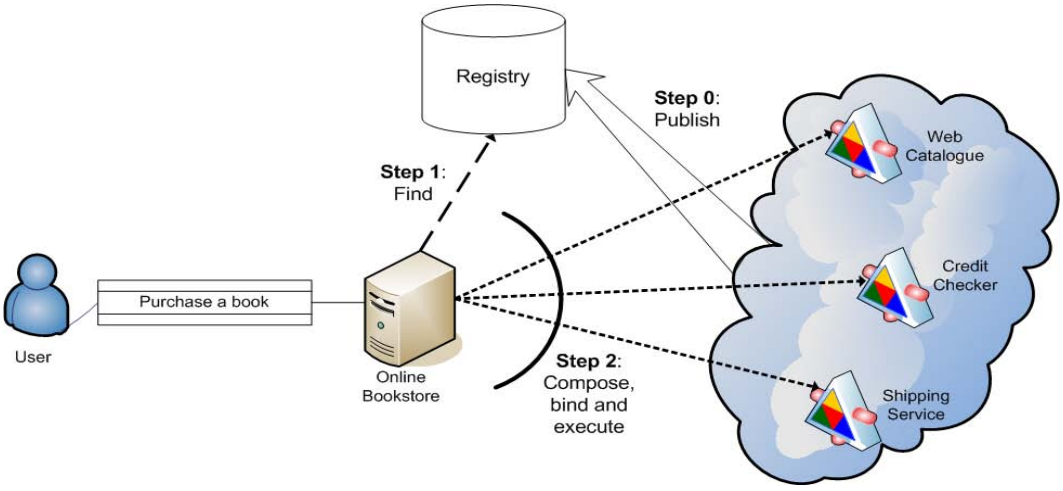

Figure 2.5: Composition of an online shopping service

An example of Web service composition is shown in Figure 2.5 and it demonstrates the online shopping domain.

In our example, the online bookstore is responsible for purchasing a book that satisfies the user request and the bookstore service itself is a composition of various other Web services, namely a Web Catalogue, a Credit Checker and a Shipping Service. The basic operation of the online bookstore service involves the following steps:

`Logon` $\rightarrow$ `Place Order` $\rightarrow$ `Check Credit` $\rightarrow$ `Pay` $\rightarrow$ `Logoff`

The user initiates the purchasing process by sending through a shopping request to the bookstore service, specifying the item(s) he/she would like to purchase. An example of the shopping request contains a shopping list that has the following attributes:

- **Item type**: Fiction book
- **Author**: Stephen King
- **Book title**: The Tommy Lockers

- **Quantity**: 1
- **Price**: $30 - $50
- **Shipping address**

Assuming all the necessary Web services have been published to the online registry (Step 0 in Figure 2.5), the bookstore service can then go to the registry and find all the relevant services for its purpose (Step 1 in Figure 2.5). After a list of relevant services has been discovery, they are composed, bind and execute together which forms the bookstore Web service (Step 2 in Figure 2.5). Once a list of relevant services has been discovered, they are composed, bound and executed together. This constitutes the bookstore Web service (Step 2 in Figure 2.5). The bookstore Web service acquires the item that is requested by the user through the Web catalogue. The Web catalogue, consisting of a set of items and information on its attributes, prices and current availability, is presented. The current availability of an item is updated during execution. As soon as such an item has been found and all the requirements have been met, it will go through the checkout process. The checkout process involves calculating the total cost, in other words the shipping cost plus the cost of the item. The shipping time and cost is determined by the Shipping Service. Lastly, the Credit Checker, one of the internal processes called `CreditApproval` and described in WSDL and OWL-S, is shown in Listing 2.1 and 2.2 respectively, and checks for sufficient funds before the payment is being processed. By mapping the abstract composition model shown in Figure 2.4 and the concrete example shown in Figure 2.5, we will see their correlations (Table 2.5).

Techniques for Web services composition are discussed in the section that follows.

Table 2.5: Correlation between abstract composition model and concrete online bookstore example

Element of the abstract composition model	**Elements of the concrete composition example**
Service requestor	User
Composite service	Bookstore service
Participating Web services	Web Catalogue, Credit Checker and Shipping Service

2.5 Current Composition Techniques

The fast-growing number of services available on the Internet results in a fast development of standards for process-based Web services composition, such as BPEL [38] and BPML [3]. Process-based Web service composition refers to composing a composite service by gluing Web services together by means of a process model. The process model specifies the composition workflow between participating services in the form of mono building blocks.

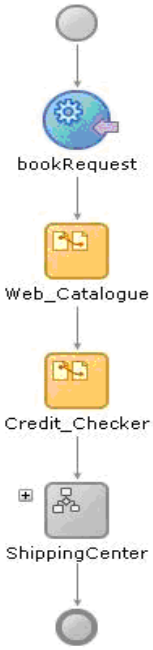

Figure 2.6: Composite service (Bookstore) in BPEL

Figure 2.6 depicts the bookstore example in BPEL. The composition process is initiated by the *bookRequest* service. It receives input from the client and based on the input, it goes to the *Web_catalogue* to check for the availability of the item specified. The *CreditChecker* checks the credit rating of the client and if approved, the *ShippingCenter* will initiate shipping to the client.

Since the BPEL composition is in a monolithic form, it lacks certain aspects such as process modularity and flexibility [1]. There is a close relationship between modularity and flexibility. While the lack of modularity hampers reusability, it will also affect flexibility. Without the ability to reuse parts of the code, the composition will be inflexible. The reason is that flexibility only comes when the composition can select parts of the code to be changed as needed.

From above, we see that the process-based Web service composition is equivalent to the proactive service composition mentioned in Section 2.4. In other words, these standards do not really address the issues of dynamic process creation. The fast pace of Internet growth causes the manual composition of Web services to become inefficient and impractical. The ability to automatically plan the composition of Web services dynamically and to monitor their execution is therefore an essential step toward the effective usage of Web services.

In order to automate the composition process, the services need to be described in a way that is unambiguous and computer-interpretable. The semantic Web was designed to facilitate the ontology-based discovery and composition of services. An example of OWL-based Web service ontology standards that annotate Web services description is OWL-S 1.1 [30]. Aslam *et al.* [4] proposed a way to map the BPEL process model the OWL-S suite of ontologies to overcome the semantic problem. Moreover, AI planning techniques have been employed to automate the composition of Web services described using these standards [48]. The following steps can be used to automate a composition process:

1. Extend current service description to incorporate semantic. For example, we could translate the existing BPEL process to OWL-S [47].

2. Translate semantic Web service description to planner actions. For example, we would like to use SHOP2 to generate a composition. We will need to encode a composite process composition problem as a SHOP2 planning problem, so SHOP2 can be used with OWL-S Web Services descriptions to automatically generate a composition of Web services calls [49].

3. Generate a composition plan using a planner and execute.

In the next section (Section 2.6), we examine the AI planning techniques and their application in Web services composition.

2.6 AI Planning

In [43] the current AI planning techniques and their application in Web services composition are surveyed. The following sections provide an introduction to general AI planning and in particular the Hierarchical Task Network (HTN) planning. The suitability of HTN planning to Web service composition will also be discussed in this section.

2.6.1 Classical Planning

A classical planning environment is an environment that is fully observable, deterministic, finite, static and discrete [23]. In general, a classic AI planning problem has the following components:

- A set of possible states of the world, S
- An initial state description, $S_0 \subset S$
- A desired goal state description, $G \subset S$
- A set of possible actions[4], A

[4]Also known as operators, the term will be use interchangeably throughout the dissertation.

- The translation relation $\Gamma \subseteq S \times A \times S$ defines the precondition and effects for the execution of each action.

The objective of AI planning is to generate a plan, that is, a sequence of actions that when executed from the initial state, will result in the desired goal state. A state is a situation that describes the world at a certain point in time. A set of ground literals in a first-order language represents a state. The actions are expressions that are described using some formal language consisting of a name, a parameter list, a precondition list and an effect list.

To understand how Web service composition can be modelled as planning, we need to know how they can be mapped up.

The initial state and the goal state of the Web service composition can be directly translated into a planning problem. In terms of Web service, action is a set of available services and the translation relation defines the precondition and effects for the execution of each action and denotes the current state of each service. The input and output parameters of the service act as knowledge preconditions and knowledge effects in a planning context. The side-effects of the service are modelled as non-knowledge preconditions. Table 2.6 shows the mapping between Web service and planning.

Table 2.6: Mapping between Web service and Planning

Web service context	**Planning context**
Web services	Actions
Current states of a Web service	Translation relation
Input parameters	Knowledge preconditions
Output parameters	Knowledge effects
Side-effects	Non-knowledge preconditions

Consider the online bookstore example (described in Section 2.4). The equivalent planning problem to this example is illustrated in Listing 2.3. To keep it simple, we will only show two services, namely find the book and the credit checker. The shipping service is therefore omitted.

```
GOAL:       (FoundBook and CreditApproval)
INIT:       (gotBook, haveCredit)
ACTION:     Book()
PRECOND:    gotBook
```

```
EFFECT:         foundBook

ACTION:         Credit()
PRECOND:        haveCredit
EFFECT:         creditApproval
```

Listing 2.3: A Planning Problem for an Online Bookstore Domain

This example describes a simple planning problem with initial state `(gotBook, haveCredit)` and the goal to satisfy `FoundBook` and `CreditApproval`. Starting from the initial state, it searches through the planning domain for first actions that satisfy the precondition. In this case only two actions: `Book()` and `Credit()` satisfy the precondition. For example, if we chose to execute `Book()`, then `FoundBook` will become true and hence be added to the plan. This also satisfies part of the final state.

The solution to this problem is a partial plan (sequence in which the actions must occur is not fully specified) that consists of two steps `Book` and `Credit`. In this example, the partial-order plan is stated as follows:

- The book can be found if and only if `gotBook` is true.
- The credit check can be approved if and only if `haveCredit` is true.
- However, the execution order of `Book` and `Credit` is not specified, but they must be true before the end of the process.

The advantage of a partial plan is that it allows least commitment, i.e. it avoids problems causing by early commitment decisions. In contrast, total-order plan is one that consists of a linear list of steps; the sequence in which the actions must occur is fully specified. Each total-order plan is a linearisation of the partial-order plan. Figure 2.7 shows both partial-order and total-order plans to the problem.

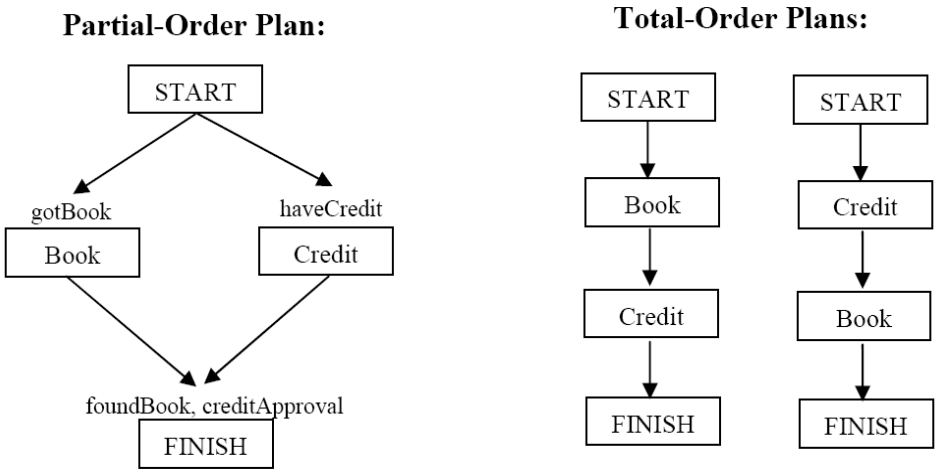

Figure 2.7: A partial-order plan for purchasing a book and the two possible linearization of the plan

2.6.2 Hierarchical Task Network (HTN) Planning

HTN planning is a technique that creates plans by task decomposition. In addition to an initial state, a goal state and a set of operators found in classical planning, HTN planning also includes a set of methods. The planning problem is specified by an initial task network, which is a collection of tasks that need to be performed under a specified set of constraints. The planning process decomposes the collection of tasks in the initial task network into smaller subtasks until the task network contains only primitive tasks. The decomposition of a task into subtasks is performed using a method from a domain description. A method specifies how to decompose the task into a set of subtasks. Each method is associated with various constraints that limit the applicability of the method to certain conditions and define the relations between the subtasks of the method. HTN planning performs a recursive search of the planning state space via task decomposition and constraint satisfaction.

Consider the online bookstore example again. By applying action, decomposition is now refined and consists of operations that find a book, check for credit approval and provide shipping details. These operations are considered to be the knowledge of how we want to implement tasks to solve

the problem. The refinement process is continued until only primitive tasks remain in the plan for each operation. A graphical representation of the refinement process for the online bookstore problem is illustrated in Figure 2.8. The node with no outgoing arrows indicates a primitive action. There are three partial-order plans, namely find a book, credit check and shipping, and the composite plan is generated by combining these three partial-order plans with task decomposition.

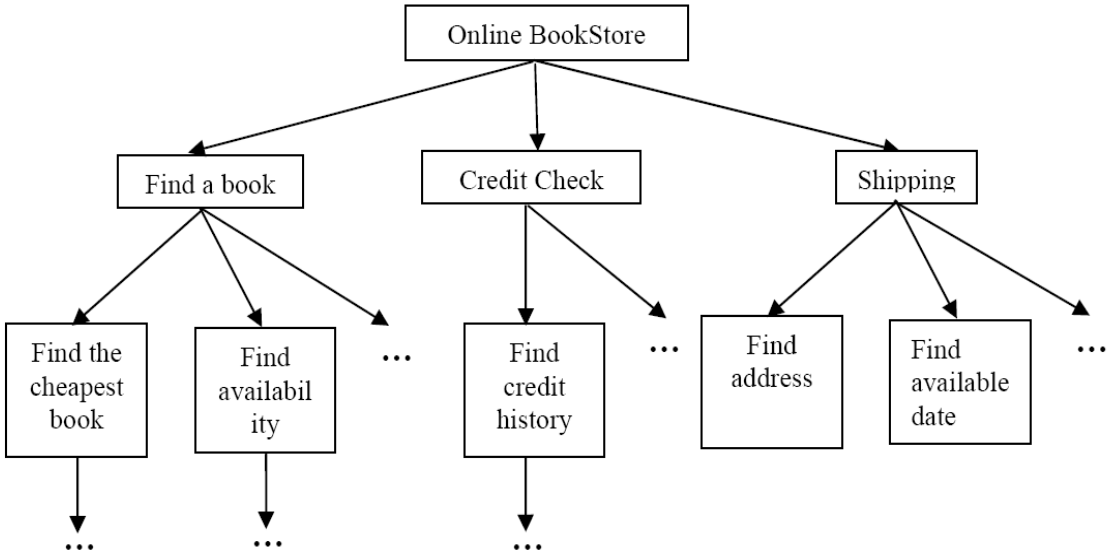

Figure 2.8: Action decomposition for an online bookstore problem

The main advantage of HTN planning is its ability to deal with very large problem domains by decomposing the goal into smaller manageable subtasks that can be solved directly. However, this technique requires experts to provide the planner with a task that it needs to accomplish. Despite this disadvantage, HTN planning is still a good way to provide different levels of abstractions. This means that the users do not need to possess complete knowledge about the services in order to select them.

2.7 Suitability of HTN Planning to Web Services Composition

Sirin *et al.* [49] mentioned various reasons why HTN planning is suitable for Web services composition. This was taken further in the current study by comparing HTN planning to other planning techniques [15]. The results show that HTN planning does indeed have various advantages over the other planning techniques in Web service composition. In this section, the two most important reasons why HTN planning is suitable for Web services composition are presented.

The concept of compound tasks in HTN planning is very similar to the concept of Web services composition. A Web services composition workflow has a complex structure with many execution paths. Information concerned with the composition can be fed to an HTN planner as a planning domain, and planner would compose a sequence of actions that would constitute a valid composition as specified using HTN methods. The designer of a method does not have to have close knowledge of how the further decompositions will go or how prior decomposition occurred. This eliminates problems such as early-commitment, which is one of the problems in the current approaches in composition. It also implies that HTN encourages modularity. Since the method author does not have to possess complete knowledge of the methods, he/she is allowed to focus on the particular level of decomposition at hand. Such modularity fits in well with Web services, meaning that methods coming from different sources can be integrated to produce suitable workflows.

Another important reason why HTN planning is suitable for Web services composition is its ability to scale well to large and complex problems. To accomplish scalability, HTN planning has the ability to scale to large numbers of methods and operators as method decompositions provide the means to prune the search space by ignoring unrelated method descriptions.

2.8 Conclusion

This chapter examined ways of describing a Web service and the way in which services can be combined into a composite service, known as Web service composition. Current composition techniques have been briefly discussed and their problems in the dynamic composition of Web services have been identified.

The application of AI planning techniques in Web service composition was touched upon and special attention was paid to the representation of planning problems in both classical planning and HTN planning. The chapter concluded with discussions on the suitability of HTN planning for Web service composition.

The next chapter explores problems encountered during Web service composition in a dynamic environment and the need for self-healing Web service composition. Moreover, a list of requirements for self-healing Web service composition is identified.

Chapter 3

Failures and Self-healing

3.1 Introduction

The previous chapter concluded that using semantic Web ontology to describe Web services is useful for automatic service composition. AI planning techniques can be used for automating Web service composition, provided that the service is described by using ontology and by representing it as a planning problem. The solution to a planning problem is known as a plan. Nevertheless, in realistic domains or entities the plan does not always execute as planned. Therefore, it is important to monitor the execution of a plan and react to unexpected events. A composition may fail either because it does not comply with the requirements and specifications, or because the specification does not adequately describe its function. An error is the part of the composition state that may cause a subsequent failure. A failure occurs when an error reaches the service interface and alters the service. A fault is the adjusted or hypothesised cause of an error. Maintaining the correctness of composition requires knowing what "correctness" is and recognising when a composition needs to be healed. The first step is to establish the criterion for correctness, which depends on the user requirements and specification. The second step is to recognise the difference between correct and incorrect conditions (i.e. failures). To react to such incorrect conditions, it is vital that we know the cause and effect of the failures.

The chapter (Section 3.2) starts off with a discussion on the cause of fail-

ures that may occur during the composition process. In describing the different faults that make up the taxonomy, examples in Oracle BPEL Process Manager Suite [39] are used. The reason for choosing Oracle BPEL Process Manager Suite for demonstration is to provide better visualation of the problem. In addition, the examples can easily be transformed into HTN operators at a later stage for experimentation purposes.

In Section 3.3, the taxonomy is presented and the most critical effects are identified. Sections 3.2 and 3.3 recount the foundations of the research and proceed to the needs for self-healing Web service composition. These needs are explained in Section 3.4. The chapter is ended with a list of requirements for self-healing composition of Web service. These requirements will be used in Chapter 4 to evaluate the applicability of HTN planners in a self-healing context.

3.2 The Cause of Failures

Languages that are being developed to detect failures in Web services can relate the effects observed to a possible cause, and then pass on this information to the recovery process [7], [9]. Different causes (faults) clearly impose different reactions, but a case-by-case decision is not feasible. The identification of classes of faults helps abstract the problem, identify generalised reactions (or reaction patterns), and also organise new faults to automatically identify the most suitable reactions. Following the work of [6], the faults are divided into three major, yet partially overlapping groups:

- **Physical faults**: include all faults arising from hardware problems, including in the network.
- **Development faults**: include all faults that can be traced to errors made during development.
- **Interaction faults**: include all faults that occur during services interaction.

3.2.1 Physical Faults

Physical faults are observed as failures in the network medium, or failures on the server side. They include communication infrastructure exceptions and failures in the correct operation of the middleware of the hosting servers. Good examples of such failures would be a server that is out of action or a severed connection to the server.

Web services are the building blocks for Web service composition. In most trivial cases, a composition fails because of the building blocks are **unavailable**. The availability of a Web service is influenced by the server and by the networking media.

There are two causes of service unavailability – either the service is down or the network connection to the service is down. It is difficult to pinpoint the exact problem but any failure that results in no response from the above two forms can be classified as an unavailability fault. This fault, and any other faults residing at the host, can only be fixed by the service provider.

3.2.2 Development Faults

Development faults may be introduced into a system by its environment, especially by human developers, development tools, and production facilities. Such development faults may contribute to partial or complete failures, or they may remain undetected until the usage phase.

A **parameter incompatibility fault** is the case of a service receiving incorrect arguments or incorrect parameter types as input. Under normal circumstances, such a fault can be avoided if catch blocks and exception handlers are used. Consider the error message in Figure 3.1, as produced by Oracle BPEL Process Manager Suite [39]. In the experiment, the service that was invoked expected an integer value as input, but received a string value instead. The service failed to be invoked since we did not give it the appropriate exception-handling mechanisms.

This type of failure can occur when using a composite service. If we dynamically compose a composite service, we might end up using two services that send incompatible types to each other. This can be handled by an

```
Message handle error.

An exception occurred while attempting to process the message
"com.collaxa.cube.engine.dispatch.message.invoke.

InvokeInstanceMessage"; the exception is: XPath expression failed to
execute.

Error while processing xpath expression, the expression is
"((bpws:getVariableData("inputVariable", "payload",
"/client:DummyService_3ProcessRequest/client:input") mod 2.0) = 1.0)",
the reason is NaN is not an integer.

    Please verify the xpath query.
```

Figure 3.1: Error produced on incorrect input

appropriate translator between the two services, but only if we are able to anticipate the incompatibility. In some cases, if a Web service is invoked using incorrect parameter types, the service will return an error message or error code that can be used to identify the type of fault that occurred, as shown in Figure 3.1. When using Web services in the .NET environment, translation tools exist that will translate the Web service's WSDL file into usable and readable program code which can be used to invoke the service. When using a Web service in such a way, it is almost impossible to cause such a fault, since the compiler will pick up on an incompatible parameter before it can be invoked.

A Web service often does not have full knowledge about services that it will interact with. Selection of the services is based on the interfaces or ontologies (descriptions) provided. Often assumptions need to be made during the selection procedure, but these assumptions can be violated. This could happen simply because the developers are too optimistic or pessimistic about their assumptions or the service update changes its interface. After the update, the **interface might have changed** and subsequent queries to the old interface would fail.

The worst-case scenario is that the interface changes, but the workflow logic stays the same. This can cause a **workflow inconsistency problem**. In the case of such a fault, the service cannot be invoked since the interface does not correspond to the workflow description. If service users are not informed about such changes, they would assume that the service is broken or no longer exists. These types of errors can occur after updating the Web

service description. Unfortunately these errors are also the hardest to pick up, since they will behave very much like a physical fault. The error might even be mistaken for a physical fault unless the service returns a sensible and usable notification of the type of error that occurred.

As an example, consider the following pieces of code (Figure 3.2 and Figure 3.3). In the example code, the BPEL code was left unchanged, but the developer changed the interface (the WSDL code). If such a change was made whilst a requestor was using this service, the service would become unavailable.

The highlighted sections of Figure 3.2 and Figure 3.3 represent the parts that were affected. The interface to the service was changed (the WSDL code) and the workflow code was left unchanged (the BPEL code). Then some of the variables' message types were changed in the interface (WSDL code). The input variable was changed from `MapServiceRequestMessage` to `MapServiceInvoked-Message`. The fault variable was also affected.

```
<partnerLinks>
    <partnerLink name="client" partnerLinkType="tns:MapService"
                 myRole="MapServiceProvider"/>
</partnerLinks>
<variables>
    <variable name="input" messageType="tns:MapServiceRequestMessage"/>
    <variable name="output"
              messageType="tns:MapServiceResponseMessage"/>
    <variable name="fault" messageType="tns:MapServiceFaultMessage"/>
  </variables>
```

Figure 3.2: BPEL code of service with workflow error

A **fault due to non-deterministic actions** is the case where a service will have more than one possible outcome or return value. Such a service might produce any one of a number of outputs that are determined by an operation that might or might not precede the output. Figure 3.4 depicts a a generic service with two identical outputs going to two different services. Such a scenario is known as non-deterministic actions.

```
<types>
  <schema>
       <element name="request" type="string"/>
       <element name="response" type="string"/>
       <element name="error" type="string" />
  </schema>
</types>

<message name="MapServiceInvokedMessage">
  <part name="payload" element="tns:request"/>
</message>
<message name="MapServiceResponseMessage">
  <part name="payload" element="tns:response"/>
</message>
<message name="MapServiceErrorMessage">
  <part name="payload" element="tns:error" />
</message>

<portType name="MapService">
  <operation name="process">
       <input message="tns:MapServiceInvokedMessage"/>
       <output message="tns:MapServiceResponseMessage"/>
       <fault name="MapNotFound" message="tns:MapServiceErrorMessage"/>
  </operation>
</portType>
```

Figure 3.3: Corresponding WSDL code of service

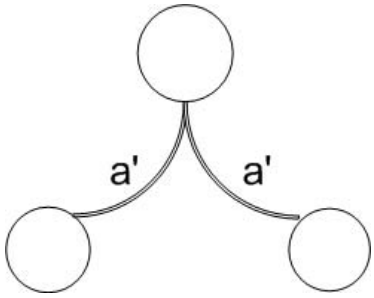

Figure 3.4: Example of Non-deterministic Actions

3.2.3 Interaction Faults

Faults from various classes can manifest and propagate from one service to another during the execution phase of the composite service. The faults may cause unacceptably degraded performance or total failure to deliver the specified service.

An interaction fault occurs when the given composite service fails more frequently or more severely than acceptable. Overall, interaction faults can be further subdivided into the following two categories:

- Content faults: include incorrect service, misunderstood behaviour, response error, QoS and SLA faults.
- Timing faults: include incorrect order, time out, misbehaving workflow faults.

Content faults occur when the content of the service delivered, based on the service description, deviates from the expected composite service. Timing faults, on the other hand, are concerned with the time of arrival or the timing of the service delivery that will result in the system deviating from its original specified functional requirement.

The integration of Web services into a composite service requires communication between all participating Web services. The communication protocol used by Web services to carry messages between each other is normally Simple Object Access Protocol (SOAP). During message-passing the message packets may arrive in a different order to the order in which they were sent. The cause of this scenario is often due to a slow network. In a composition, services are often dependent on each other to reach a desired result. In the case where a message arrives in a different order it may cause undesirable results. A possible solution to this problem is to make use of Lamport timestamps [28].

Another possible fault caused by a slow network is a **time-out** exception. If proper measures are taken, time-out exceptions and communication medium failures can be caught in time, using exception handlers. For example, catch blocks and exception handlers in BPEL can be used to recover from

certain failures and exceptions. Baresi *et al.* [8] give an example in which they make use of the appropriate exception handlers to catch a time-out exception.

There are several classes of faults that form part of the interaction fault category, one of which is **misbehaving execution flow**. Dynamic service composition means creating a composite service on the fly. Predefined interaction is not possible and it varies according to the dynamic situation. This increases the probability of a fault, because a single service may have several dependencies that are stimulated to correctly perform their tasks, but the developers may not know the identity of the services that will fulfil the request. In the same way, the service that will be used to satisfy a dependency may not be aware of this fact until deployment. To better understand a misbehaving execution flow fault, consider the example shown in Figure 3.5. The composite service is defined by its interface and the internal workflow of the composition that is contributed by participating Web services $(w_a, \ldots, w_z)$. A misbehaving workflow fault may therefore be caused by one of the following:

- The workflow of a composite service is not correct. That is, the result returned is not what was expected or the service returns no results at all.
- An individual service (e.g. w_c) used by a composite service is not correct.
- An individual service (e.g. w_f) used by a composite service does not work well with other participating services.

A **misunderstood behaviour fault** is the case of a service requestor that requests a service from a service provider, expecting a service different from the provided one. A simple example would be if the requestor requests a service for stock exchange quotes, and the provider returns a service supplying exchange rate quotes. These types of faults can occur if the description of a service is incorrect, or if the service provider misinterpreted the request from the requestor.

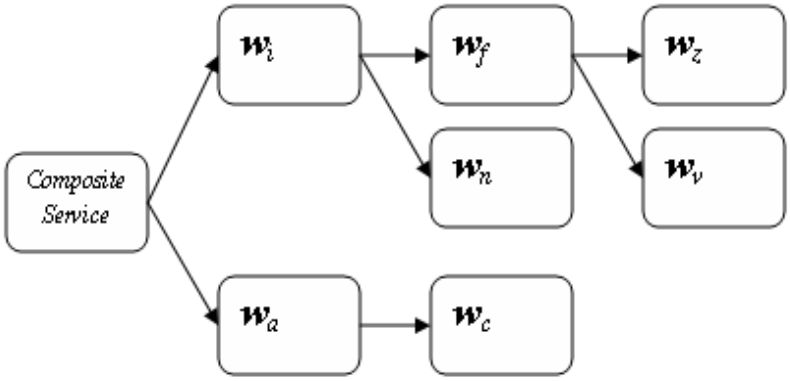

Figure 3.5: Composite service example

Response faults are closely related to behaviour faults and incompatible input. A service that exhibits this type of failure will sometimes produce incorrect results even if the correct input was received. The incorrect results will be caused by the incorrect internal logic of the service. Some faulty services might even randomly produce outputs for a requestor. In this case, it would be better to rebind to a different service. While response faults generate wrong values, they are often due to the misunderstanding of service behaviour and wrong input values.

Non-functional aspects of Web services include the **Service Level Agreement (SLA)** and the **Quality of Service (QoS)** agreement. These non-functional properties can be captured at runtime and used to monitor the performance of the service with respect to the contract between different parties.

According to Ludwig *et al.* [11], SLAs are used for the reservation of service capacity. They were traditionally used only between organisations to reserve the use of services between them. SLAs have only recently been used as a way to ensure delivery of services in Web services and SOA. A fault caused by or during the SLA will manifest in the requestor receiving an incorrect service. During SLA negotiations all parties involved will have to agree on the interface and the service provided by the provider. If the provider is providing a service that was not originally promised or agreed

upon, then we have an SLA disagreement fault.

QoS includes not only SLAs, but also the promise to deliver quality service in terms of speed and information. If any of these factors are not of good quality (i.e. if the speed is slow or if there are delays in the responses from the service) then we have a fault caused by the QoS factors. There are ways to use QoS constraints to ensure that a service can deliver what is promised. Yu and Lin [56] propose a method for binding to a service by making use of QoS constraints as a heuristic to choose the service. Faults caused by these and other non-functional aspects can only be caught after execution and use of the service, and the best way to fix them would be to rebind to a different service that promises to offer a better quality of service.

An **incorrect service fault** is closely related to SLA and QoS faults. It occurs when the provider provides a service under false pretences. It does not mean that the service is not working. The service may be working and may deliver the best results yet, but it is not the service that you requested. This can happen if the ontological description of the service is incorrect. It can also happen if the WSDL description of the service is incorrect. Most of the time, this fault goes hand in hand with SLA and QoS faults.

3.3 Fault Taxonomy

The preceding section provided an overview of faults and the observed effects in a composition process. A summary of our findings is presented in Table 3.1. The three broad categories of faults are shown along the top of the table, broken down into the subcategories identified in Section 3.2. Alongside the table six fault classes are shown that are relevant for Web service, based on the sixteen elementary fault classes identified Avižienis *et al.* [6].

Phase of occurrence:

Development faults occur during system development and maintenance. They may be introduced into the system by its environment, especially by human developers, development tools, and production facilities.

Such development faults may contribute to partial or complete failures, and they may remain undetected until the usage phase.

Operational faults occur during service delivery or during use of the Web service. These types of faults can be traced back to any one of a wide array of causes. Sometimes the cause lies at the server's side and can only be fixed by the service developer. In a few cases, the cause can be traced to the client's side. In such a case it can be fixed easily by fixing the underlying problem that caused the failure in the first place.

Table 3.1: The classes of the combined faults

		FAULTS												
		Physical	Development				Interaction							
		Unavailability	Parameter Incompatibility	Interface Change	Workflow Inconsistency	Non-Deterministic Actions	Incorrect Order	Time Out	Misbehaving Flow	Misunderstood Behaviour	Response Error	Service Level Agreement	Quality of Service	Incorrect service
FAULTS	Development		X	X	X	X								
	Operational	X					X	X	X	X	X	X	X	X
	Internal								X			X	X	
	External	X	X	X	X	X	X	X		X	X			X
	Hardware	X					X	X						
	Software		X	X	X	X			X	X	X	X	X	X

System boundaries:

Internal faults originate inside a single service. Some of the most common faults classified as internal faults are interface change, workflow inconsistency and non-deterministic actions.

External faults originate outside a single service and are propagated into the composition by interaction or interference. These include all types of hardware faults and input faults. A good example of this type of fault would be a parameter incompatibility fault.

Dimension:

Hardware faults originate in or affect the hardware. This can include a broken communication medium, a server down time, or even hardware faults on the client's side.

Software faults are caused by mistakes made in the code of the Web service. These faults occur during development or maintenance of the service and can only be fixed by the developer of the service. It also includes composition faults, if the composition is done on the client's side. In such a case, the composition can be fixed on the client's side.

Figure 3.6 presents an alternative representation of our taxonomy, along with the observed effects shown at the bottom of the matrix. The observed effects were obtained through various testing and experiments. From the matrix it can further be observed that the majority of faults lead to the following outcomes, namely unresponsive Web service, incorrect results, incoherent results and slow service. These outcomes, together with the fault classes from where they originated, are shown in Table 3.2. Since the outcomes originated mostly from development and interaction faults, it can be concluded that the most critical faults are development and interaction faults.

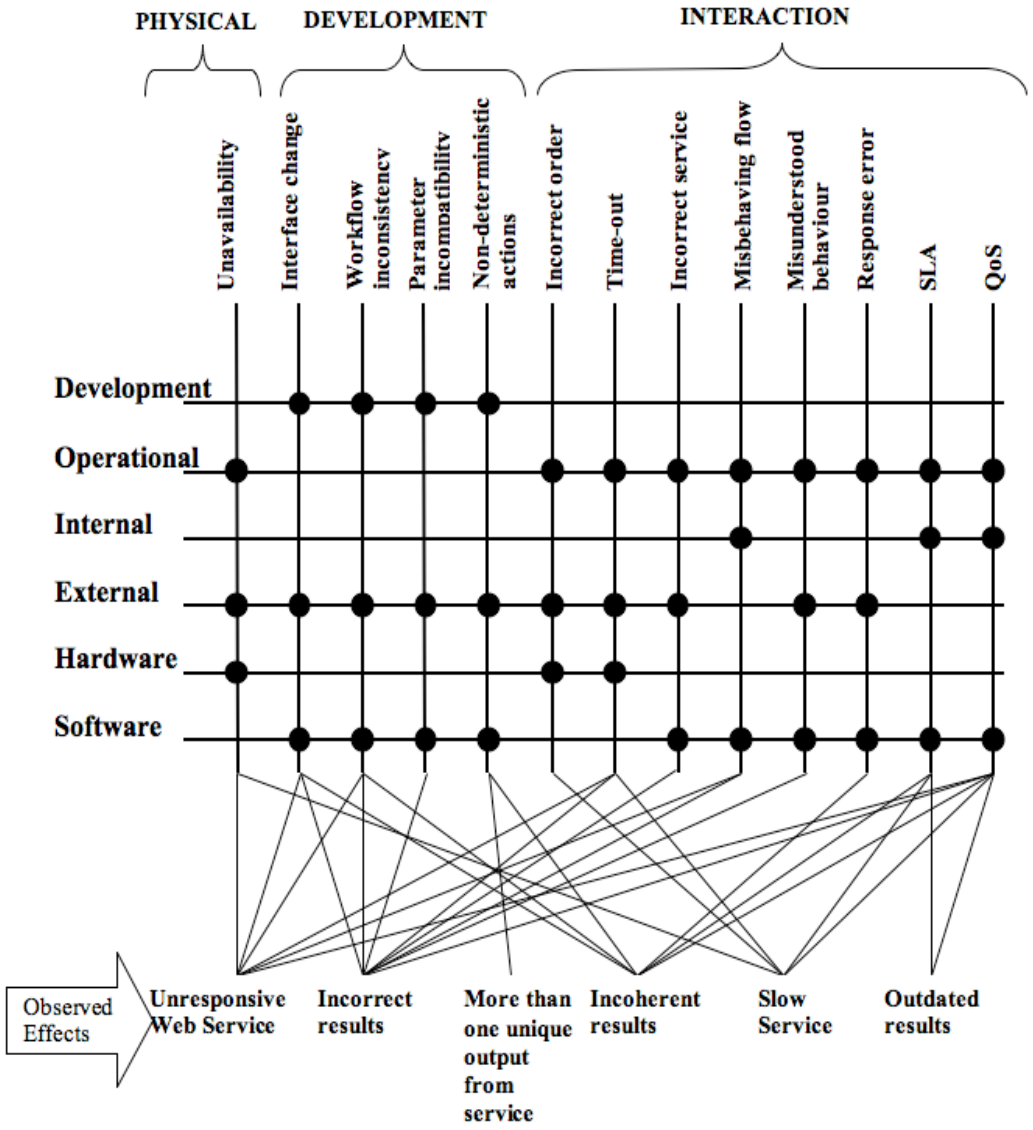

Figure 3.6: Matrix representation of the combined classes and observed effects

Table 3.2: Outcomes and their origin

Outcomes	**Main fault categories**	**Fault subcategories**	**Elementary fault classes**
Unresponsive Web service	- Physical - Development - Interaction	- Unavailability - Interface change - Workflow inconsistency - Time-out - Misbehaving flow - QoS	- Operational - External - Software
Incorrect results	- Development - Interaction	- Interface change - Workflow inconsistency - Parameter incompatibility - Time-out - Incorrect service - Misbehaving flow - Misunderstood behaviour - QoS	- Operational - External - Software
Incoherent results	- Development - Interaction	- Interface change - Workflow inconsistency - Non-deterministic actions - Response error - SLA - QoS	- Development - Operational - External - Software
Slow service	- Physical - Interaction	- Unavailability - Incorrect order - Time-out - SLA - QoS	- Operational - Internal - External - Hardware - Software

3.4 Requirements

Web service composition depends on specifications defined by users, but it is often impractical to obtain precise specifications of the desired composite service. Most users are inarticulate about their criteria for correctness, performance and Quality of Service (QoS). Furthermore, the criteria for acceptable behaviour vary from time to time and from one user to another. The need for self-healing arises because the composition may be subject to unpredictable external requirements and/or it may be too complex to predict its internal behaviours precisely. In this study self-healing Web service composition is defined as follows:

"A Web service composition process that has the ability to heal itself from errors and exceptions in dynamic and uncertain environments."

In designing a cycle for self-healing Web service composition, it is wise to first ask what the requirements for such a cycle should be. If such a list can be complied, then a base can be established against which implementation can be achieved in the development of self-healing Web service composition.

Below is a list of requirements that are essential in self-healing Web service composition. These requirements have been distilled from numerous sources by evaluating current approaches in self-healing systems [54], [36], [35], [40], [26], [25], [22], [21], [29]. The self-healing Web service composition cycle is shown in Figure 3.7. It is comprised of three modules, namely **Plan Generation Module**, **Plan Execution Module** and **Failure Analysis Module**. These proceed in several steps, shown in Figure 3.7.

The user sends the request to the plan generation module (Step 1). The request is then passed by the request parser (Step 2) to the planning system to generate a plan (Step 3). During the plan generation process, verification is undertaken to ensure the quality and correctness of the plan generated. The verification process includes checking that the plan generated matches the request that was sent by the user, as well as checking all the constraints that need to be satisfied.

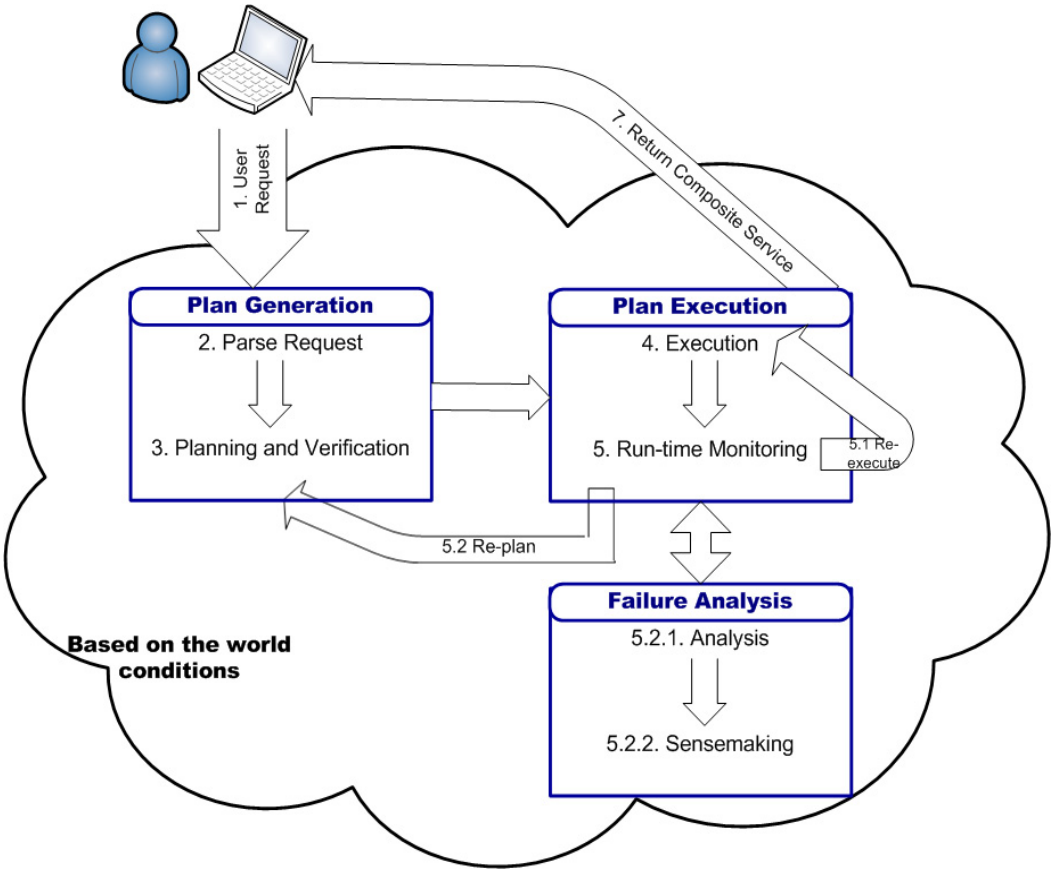

Figure 3.7: Self-healing Web Service Composition Cycle

After a plan has been successfully generated, it is passed to the plan execution module for real-time execution in the dynamic environment. During the execution phase, the plan is monitored by a run-time monitor (Step 5) to ensure the plan is executed as expected. If, at any stage during the execution, the monitor detects some unexpected occurrences, it will send them to the Failure Analysis Module to analyse the exact problem and its cause (Step 5.2.1). In the event where the problem is not recognised, a Sensemaker (Step 5.2.2) is used to reason about the problem, based on the current world condition, in order to come up with a reasonable description of the problem.

In our self-healing cycle, sensemaking is the process of creating situation awareness and understanding, in run-time, situations of high complexity and uncertainty in order to come up with a description of the problem. After

the problem has been correctly identified, there are two ways to overcome it, based on its severity:

1. Re-execute the original plan from the current state with the belief that the environment will return to a more conducive state (Step 5.1).

2. Re-plan from the current state to incorporate the latest available information and develop a partially new or completely new plan. Then, re-execute (Step 5.2).

Now consider a more detailed description of the requirements identified.

- ***Run-time monitoring*** of the composition checks consistency of the specification. Based on the user request specification, the run-time monitor can verify and detect run-time faults/errors (Step 5).

- ***Analysis*** is used to identify faults/errors according to the fault taxonomy and to provide a recommendation (Step 5.2.1).

- ***Sensemaking*** is required to reason over the faults/errors that are not recognised by the analysis step (Step 5.2.2).

- ***Planning and verification*** change the plan according to the faults/errors detected. The output can be a new plan or most often an alternated plan. The plan that is generated is then verified for correctness. The verification can either be done at each step during the plan generation or it can be done once the complete plan has been generated (Step 3).

- ***Execution*** of the plan generated by the planner (Step 4).

3.5 Conclusion

In this chapter the Web service composition problems were examined by providing the background against which self-healing Web service composition is needed.

Three main categories of service composition faults were identified, namely physical faults, development faults and interaction faults. Each of these categories was further mapped into six elementary faults. It was concluded that the most critical faults mostly originated from development and interaction faults. Based on these observations, a list of requirements for self-healing Web service composition was drawn up and explained. The requirements included monitoring, analysis, sensemaking, planning and verification, and lastly, execution.

In the next chapter we intend to determine the applicability of HTN planners against the requirements identified.

Chapter 4

Evaluation of HTN Planners

4.1 Introduction

In the preceding chapter, possible faults were identified during the execution of a composite service that would alter its behaviour. A fault taxonomy was drawn up to identify these faults. The need for self-healing Web service composition was explained and a list of requirements was drawn up. In this chapter, we elucidate how these requirements can be achieved using features provided by HTN planning systems.

In Section 4.2, several HTN planning systems are presented. The evaluation of their strengths and weaknesses forms the basis for the final assessment of the requirements identified for self-healing Web service composition. The result of the analysis is used to match up the list of requirements identified in Section 3.4 and our findings are presented in Section 4.3.

4.2 HTN Planning Systems

As was remarked in Section 2.6.2, HTN planning is a technique that creates plans by task decomposition. A variety of systems, such as SIPE-2 [52], [53], O-Plan [17] and JSHOP2 [27] have been implemented based on the HTN planning technique. An overview of these planning systems is given in the subsequent sections, followed by an evaluation of their features with respect to the self-healing requirements identified. The self-healing requirements

involved are monitoring, analysis, sensemaking, planning and verification, and execution. The evaluation is aimed at verifying the applicability of HTN planning systems in self-healing Web service composition.

4.2.1 SIPE-2

4.2.1.1 Introduction

The System for Interactive Planning and Execution Monitoring (SIPE-2) [53] was developed by the AI centre at the SRI International and is based on its predecessor SIPE [52]. The underlying techniques for the system are its support for resources and temporal constraints that allow for execution monitoring, user intervention and replanning.

The main improvement of SIPE-2 from SIPE is that it has the ability to reason about the arbitrary ordering of actions. This means that SIPE-2, if provided with an arbitrary initial situation and a set of goals, will be able to generate solution plans automatically or under interactive control by combining operators. SIPE-2 performs planning hierarchically at different levels of abstraction. Formalisms for describing actions as operators are provided at each decomposition level. At each level, the planner needs to predict how the world will change as actions are performed. The SIPE-2 architecture is capable of efficiently reasoning about actions in order to generate a novel sequence of actions that corresponds precisely to the situation at hand. In addition, it will use as much as possible of the old plan when new situations arise and can perform replanning during execution.

The following paragraphs explain the features that are offered by SIPE and focus on its execution monitor and replanner module (shown in Figure 4.1).

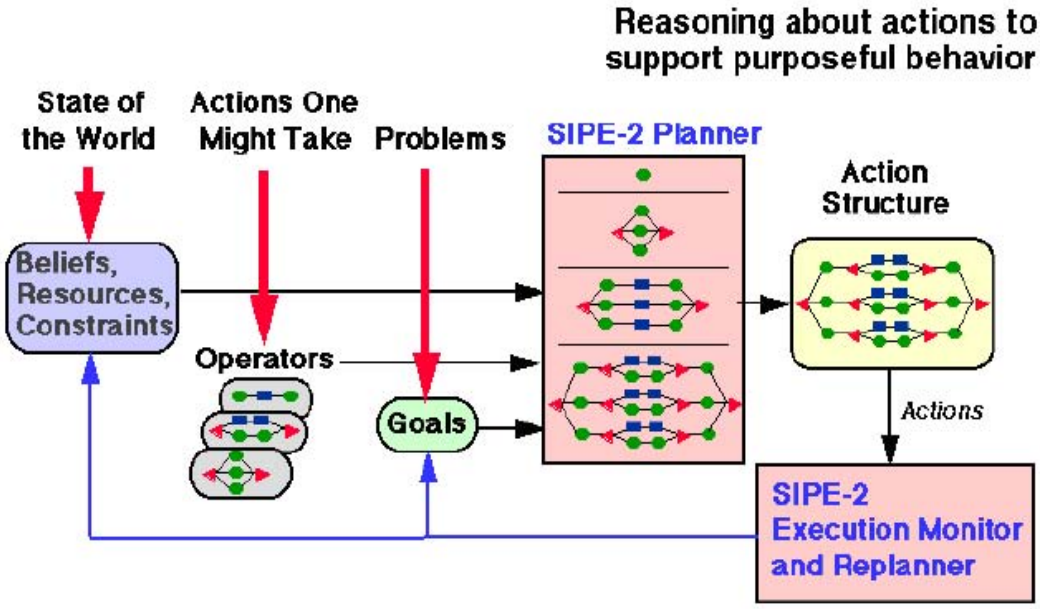

Figure 4.1: SIPE-2 Architecture [51]

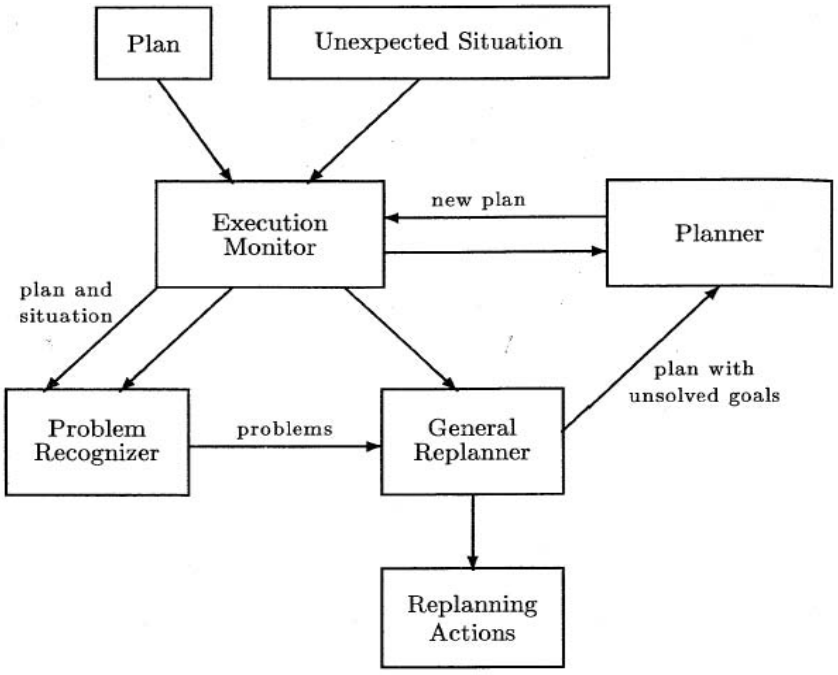

Figure 4.2: SIPE Modules and Flow of Control [52]

Figure 4.2 depicts the various modules in the SIPE execution-monitoring system. At any point during execution, the execution monitor will accept two types of information about the domain: 1) the plan that is being executed; and 2) unexpected situation(s). SIPE first checks whether the "unexpected situation" is really "unexpected". It checks whether the situation differs from its expectations and if it does, then replanning is needed.

Once the description of the unexpected situation(s) has been gathered, the execution monitor calls the problem recogniser. It provides the problem recogniser with the plan and the unexpected situation(s), and the recogniser returns a list of all the problems it has detected in the plan. The general replanner is then given a list of the problems found by the problem recogniser and it tries certain replanning actions in various cases. However, it will not always find a solution.

The general replanner changes the plan so that the latter will look like an unsolved problem to the standard planner in SIPE. Once the replanner has dealt with all the problems that were found, the planner is called and the plan, which now includes unsolved goals, is send through. If the planner produces a new plan, this new plan should solve correctly all the problems that were found. Next the plan is given to the execution monitor for continuing execution. If not, the whole cycle is repeated.

4.2.1.2 Evaluation

The requirements for the SIPE-2 planning system self-healing Web service composition are summarised in Table 4.1. The first column lists the self-healing requirements belonging to the Web service composition cycle under discussion. The second column represents the corresponding modules of the planning system in terms of those requirements.

The attributes chosen by SIPE-2 and the ways in which they conform to each identified self-healing requirement are discussed in the text that follows.

Table 4.1: Mapping between self-healing requirements and SIPE-2's replanner modules

Self-healing requirements	**SIPE-2 module**
Planning and verification	Planner
Run-time monitoring	Execution monitor General replanner Replanning actions
Analysis	Problem recognizer
Sensemaking	Problem recognizer
Execution	Planner Execution monitor

Planning and verification: To generate a plan, the SIPE-2 planner takes in four inputs, namely a description of the initial state, a description of a set of actions, a problem descriptor and a set of rules describing the problem domain. All these inputs provide the planner with the current state of the world.

The description of the initial state is encoded with a sort hierarchy, a world model and a set of deductive rules. The sort hierarchy is the order sort logic that has the ability to represent the hierarchy of the concept. For example, we understand the meaning of *'blue'*. The meaning is decided by the meaning of *'colour'* which is more general than *'blue'* in the sort hierarchy. In the SIPE-2 planner, the sort hierarchy represents the invariant predicate instance of objects, it describes the classes to which an object belongs and it allows inheritance of properties. The world model is a set of predicates that hold over objects in the sort hierarchy. Some predicates are given explicitly in the database, while others are deduced by applying deductive rules to the database.

An operator is the representation of actions, at different levels of abstraction, which may be performed in the given domain. An operator contains information about the objects that participate in the actions, the constraints on the objects in the action, a set of instructions for performing the action to achieve a goal, and the main effects accomplished by the action.

The SIPE-2 planner produces a plan through instantiation of operators

by binding their variables. It combines these instantiations by ordering them, and then adds additional constraints to avoid problematic interactions between actions. A set of operators for each goal in an abstract plan is chosen to produce a more detailed plan. To this plan the planner adds deductions and orderings that avoid problems it has detected. Thus, it provides verification on the plan that is being generated. The plan generated is represented by a partially ordered set of primitive actions and conditions that are to be carried out.

Run-time monitoring: The execution monitor and replanner are responsible for run-time monitoring. At any given time, the execution monitor can accept new information about the world and the plan that is being executed. If some unexpected events occurred, the replanning module will determine how these unexpected events have affected the plan and will modify it accordingly. As far as analysis is concerned, the problem recogniser is responsible for analysing the problems. To first recognise the problem, a 'Mother-Nature' (MN) node is inserted into the plan after the action that is presumed to bring about the unexpected effects. The purpose of the MN node is to help determine how the remainder of the plan is being affected based on its effects. Two aspects are involved in analysing the problem: the first involves finding the problem on the current plan and the second involves deductions on the world state, all based on the effects of the MN node.

Given the way the plan was produced, there are only six problems that need to be checked by the problem recogniser (Table 4.2). After the problem has been recognised by the problem recogniser, SIPE-2 offers eight replanning actions that can be used to alter the plan. The replanning actions Reinstantiate, Insert, Insert-conditional, Retry, Redo, Insert-parallel and Pop-redo are used to solve problems found by the problem recogniser, while Pop-remove is used to take full advantage of serendipitous effects.

The general replanner takes a list of problems as well as possible serendipitous effects from the problem recogniser, and calls one or more of the replanning actions in an attempt to solve each problem. The corresponding problems and replanning actions used by the replanner are shown in Table 4.3.

Table 4.2: Problems to be checked by SIPE-2 problem recogniser

Problem	**Explanation**
Purpose not achieved	Problem occurs if the MN node negates any of the main effects of the action just executed.
Previous phantoms not maintained	Problem occurs if the MN node negates a list of phantom nodes that occur before the current execution point and whose protect-until slot requires their truth to be maintained.
Process node using unknown variable as argument	If a variable has been declared unknown, then the first action using it as an argument must be preceded by a perception action for determining the value of the variable. Otherwise problems will occur.
Future phantoms no longer true	A phantom node[5] may no longer be true after the current execution point. It must be changed to a goal node so that it can be solve by the planner.
Future precondition no longer true	Problem occurs if a precondition node is no longer true after the current execution point.
Parallel post-condition not true	Problem occurs when if any of the parallel post-conditions is not true at a join node.

[5]A phantom node is similar to a goal node except that it is already true in the situation represented by its location in the plan. A phantom node is part of the plan because it may become a goal node if the plan that precedes it changes. Therefore, the truth of a phantom node must be monitored as the plan is being executed.

Table 4.3: Current Algorithm of General Replanner (adapted from [52])

Problem	**General Re-planner's response**
Purpose not achieved	Redo
Previous phantom untrue	Re-instantiate, then Retry
Unknown variable	Insert-conditional
Future phantom untrue	Retry
Precondition untrue	Re-instantiate, the Pop-redo
Parallel post-condition untrue	Insert-parallel

Sensemaking: Since the problem recogniser has the ability to recognise problems in the plan and only six problems need to be checked, there will not be a situation where a problem cannot be recognised. Therefore, sensemaking is not necessary. However, at the same time, one can argue that sensemaking and problem analyser co-exist as part of the problem recogniser.

Execution: After the general replanner has applied the appropriate re-planning actions to the problems in the current plan, it sends to the planner a new unresolved problem on which to perform planning. Once the plan is ready, the planner gives it to the execution monitor for continued execution and, at the same time, to monitor the execution of this *'new plan'*.

4.2.2 O-Plan

4.2.2.1 Introduction

The O-Plan (Open Planning Architecture) [17] project explores the issues of coordinated command, planning and control. The objective of the O-Plan project is to develop an architecture with which different agents have task assignment, planning and execution monitoring roles. The O-Plan comprises three distinct components, each with distinct responsibilities (as shown in Figure 4.3):

1. A *User Interface* that takes in the domain specification, initial state and the goal state from the user.

2. A *Plan Generator* that is responsible for the plan generation.

3. An *Execution System* that consists of an execution monitor used to gather information from the sensors or reports by the agents participating in the operation, in order to gain knowledge about the current state of the world. The information gathered is used to detect problems, to determine their effects on the plan and hence to initiate a plan repair process.

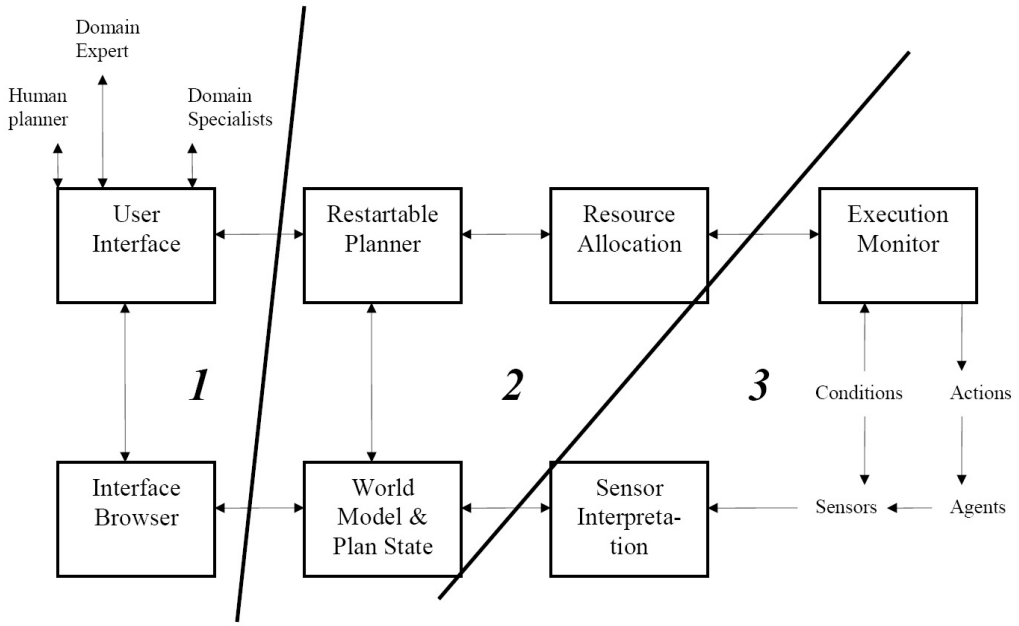

Figure 4.3: O-Plan overview [17]

O-Plan uses schemas to represent task decompositions. The Plan Generator receives a specific set of tasks to generate a plan. It then recursively expands the high-level tasks, instantiating variables as it goes, into a set of subtasks that will accomplish those tasks. A plan is then generated by choosing suitable expansions for tasks in the plan and by ordering actions and variable constraints that satisfy (and continue to satisfy) the conditions of the effects and the use of other actions.

The following example code shows an online bookstore (ref. Section 2.4) that is described by using schemas.

```
types item = (Textbook Fictionbook);
types author = (Bishop King);
types title = (CSharp TheEyesOfTheDragon);
resource_units qty = count;
resource_units price = count;
resource_units dollars = count;
resource_types consumable_strictly {resource book (item Textbook)
                (author Bishop) (title CSharp) (qty 5) (price 80)};
resource_types consumable_strictly {resource book (item Fictionbook)
        (author King) (title TheEyesoftheDragon) (qty 10) (price 50)};
resource_types
  consumable_strictly {resource fund} = dollars;

task buy_book;
   nodes
         1 start,
         2 finish,
         3 action {purchase book};
   orderings 1 ---> 3, 3 ---> 2;
end_task;

schema purchase;
   expands {purchase book};
   nodes
      1 action {find book},
      2 action {transfer fund},
      3 action {purchase book};

   conditions
      supervised {book found} at 2 from [1]
      supervised {fund transfered} at 3 from [2]
      supervised {book purcahsed} at 4 from [3]
end_schema;

;;; Primitives
schema find;
   expands {find book};
   only_use_for_effects {book found} = true;
end_schema;
schema transfer;
   expands {transfer fund};
   only_use_for_effects {fund transfered} = true;
end_schema;
schema purchase;
   expands {purchase book};
   only_use_for_effects {book purchased} = true;
end_schema;
```

Listing 4.1: An online bookstore example in O-Plan

Lines 1-12 define the variables to be used for the expansion of the task `buy_book` (line 14). The resource that is defined `consumable_strickly` means that the set amount of the resource cannot be topped up.

The task `buy_book` is comprised of three nodes, and the order of execution is defined as `start` → `purchase book` → `finish`. The purchase book action is expanded using the `purchase` schema, which in turn uses three primitive schemas, namely `find`, `transfer` and `purchase` to aid its own expansion. The output of the plan generated is shown in Figure 4.4.

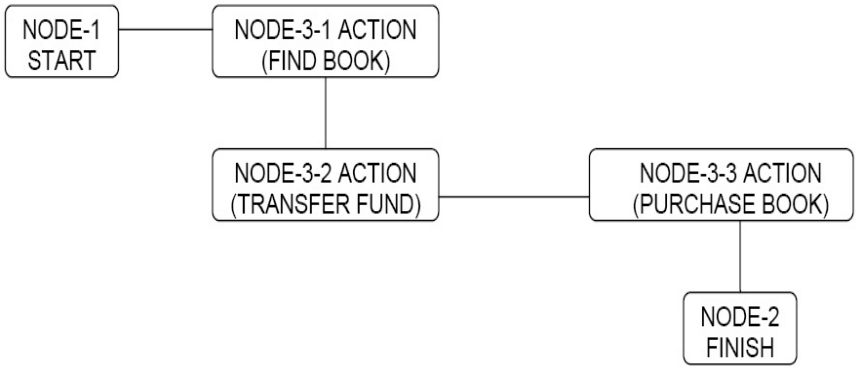

Figure 4.4: Solution to the online bookstore problem in O-Plan

During the execution phase of a plan, the execution monitor is used to gather information from sensors or reports by the agents that are participating in the operation, in order to gain knowledge about the current state of the world. The information gathered is used to detect problems, to determine their effects on the plan and hence to initiate a plan repair process.

When the execution monitor detects a problem, it sends a report to the plan generator to indicate which actions in the plan have already been completed and what other events have occurred. A search is then performed to check which effects of the action have been damaged by the event and what needs to be reinstated.

A number of plan repair mechanisms are possible, depending on the current situation:

1. Identify those parts of the plan that are intimidated by the changes in the current situation;

2. Integrate subplan requirements into the plan; or

3. Generate a completely new plan.

In particular, there are two types of problems, namely Execution Failure and Unexpected World Event, that are dealt by the O-Plan plan repair algorithm [19]. The plan representation generated by O-Plan contains two tables used by the plan repair algorithms to determine the consequences of failures, namely the **Table Of Multiple Effects** (TOME) and the **Goal Structure Table** (GOST). The plan contains nodes and each node has effects attached to it. Effects can take place at either end of a node and they are recorded in the TOME. The dependency of an effect on the node asserted is recorded in the GOST.

An execution failure occurs when one or more of the expected effects at the node-end fail to be asserted. This type of failure may cause problems if the expected effects of the action are needed to satisfy the preconditions of a later action. For example, if the ***begin_of node 1-2*** from ***end_of 1*** (depicted in Figure 4.5) fails to assert the condition as promised, then it may cause those nodes that follow to produce a result that differs from the one expected. To resolve such execution failure, an algorithm has been developed, shown in Listing 4.2. The algorithm is used by the planning system to track execution and initiate repairs, and it works as follows:

- Mark the node-end as having been executed.

- If Action (***A***) contains failed effects (***e***), then remove (***RMV***) the corresponding TOME entries. If there are no failed effects, then no repair is needed. Thus it can skip all the operations and jump to the end.

- Determine those GOST entries (***G***) that are affected by ***e***. If there are none, then no repair is needed and it can skip all the operations and jump to the end. If ***G*** exists, then search through the affected GOST entries in turn and check the validity of the contributors.

- If after the validation process the contributors are still valid, the reduce the contributors list, otherwise record GOST entry is "truly broken" (***g***). If there is no truly broken ***G***, then the repair is completed.
- For each truly broken ***g***, post a KS-FIX[6] agenda entry. When the agenda entry is processed, the KS-FIX knowledge source will be invoked, and it will choose and apply the most satisfying repair method to the GOST entry:
  - Find an existing alternative contributor in the plan.
  - Introduce a repair plan that asserts the appropriate effect. Any new nodes will be linked after the net point (highlighted in Figure 4.5).
  - Post a KS-CONTINUE-EXECUTION to continue execution after the fixes have been made.

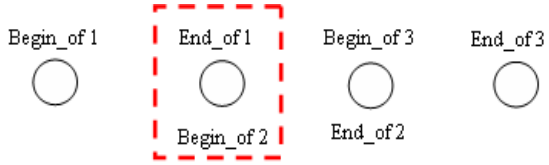

Figure 4.5: Nodes Relationship

```
Precondition: Mark the node–end as having been executed

begin
    if A contains e then
        RMV corresponding TOME entries
        if G then
            search g
                HAS !(valid) contributor
                    g is truly broken
                        post KS–FIX agenda
                        invoke KS–FIX knowledge source
                        post KS–CONTINUE–EXECUTION
                        JUMP end
                HAS valid contributor
                    JUMP end
```

[6]For more details on Planning Knolwedge Source see O-Plan Architecture Guide [50]

```
            JUMP end
17      JUMP end
end
```

Listing 4.2: O-Plan Plan Repair Algorithm for Dealing with Execution Failure

Unexpected world events are defined as events that are not in the plan that causes the planned actions to fail. They can be resolved by using the algorithm shown in Listing 4.3, which works as follows:

- Add an Event Node (***E***) and mark it as having been already executed (Figure 4.6).
- If there are any contributors who can no longer contribute, then remove (***RMV***) them from the GOST and get a list of ***g***.
- A contributor is removed in the following cases:
  - The condition is at a node-end that has not been executed.
  - The contributor is a node-end that has been executed.
  - The unexpected world event has a conflicting effect.
- For each ***g***, post KS-FIX agenda. When the agenda entry is processed, the KS-FIX knowledge source will be invoked, using ***end_of E*** as a neck point.
- Add ***e*** at ***end_of E***.
- If there is no ***g***, that means a resolution has been reached. Else, post KS-CONTINUE-EXECUTION to continue execution after the fixes have been made.

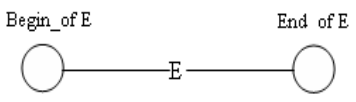

Figure 4.6: Linking of the Event Node

```
Add E
Mark E as having already been executed

begin
    if !(contribute) then
        RMV contributor
        GET g.list()

    foreach g
        post KS-FIX agenda
        add e to node-end E
    post KS-CONTINUE-EXECUTION
end
```

Listing 4.3: O-Plan Plan Repair Algorithm for Dealing with Unexpected World Events

4.2.2.2 Evaluation

The requirements for the O-Plan planning system self-healing Web service composition is summarised in Table 4.4. The table presented is to be interpreted in the same way as Table 4.1.

Table 4.4: Mapping between self-healing requirements and O-Plan module

Self-healing requirements	**O-Plan**
Planning and verification	Plan generator
Run-time monitoring	Execution monitor Plan repair algorithms
Analysis	Knowledge sources
Sensemaking	Knowledge sources
Execution	Plan generator Execution monitor

Planning and verification: When the O-Plan is given a specific set of tasks to generate a plan, it recursively expands the high-level tasks into a set of subtasks which will accomplish those tasks. Plans are generated by choosing suitable expansions for tasks in the plan and by including the set of more detailed subtasks described by the chosen expansion. The ordering of the subtasks and variable constraints are then satisfied to ensure that the

asserted effects of the actions satisfy (and continue to satisfy) conditions on the use of other actions. The expansion and instantiation occur according to a backtracking algorithm, returning the first plan that satisfies the problem constraints. By ensuring that the constraints are satisfied, the planner can ensure the correctness of the plan that is being generated.

Run-time monitoring: During execution of the plan, the execution monitor gathers information from the sensors or from the agents to determine the current state of the world. The information gathered is used to detect problems, to determine their effects on the plan, and hence to initiate plan repair. For each failing effect that is necessary for some other action to execute, additional actions, in the form of a repair plan, are added to the plan.

Analysis: The O-Plan uses the knowledge sources to determine the consequences of unexpected events or of actions that do not execute as intended. It is also responsible for making decisions on what actions are taken when a problem is detected, and for making repairs to the affected plan. However, there is no explanation on how a failure is represented.

Sensemaking: Since there is no explanation of a failure, all the reasoning of unexpected events and failures is done through the knowledge sources. In this case, the analysis and sensemaking can be viewed as a single operation.

Execution: After the repairing algorithm has been applied, the execution is resumed and monitored by the execution monitor.

4.2.3 JSHOP2

4.2.3.1 Introduction

JSHOP2 [27] is a Java implementation of SHOP2 (Simple Hierarchical Ordered Planner 2) [37]. JSHOP2 is a domain-independent HTN planning system based on ordered task decomposition. Ordered task decomposition means that the planning system plans tasks in the same order that they will later be executed.

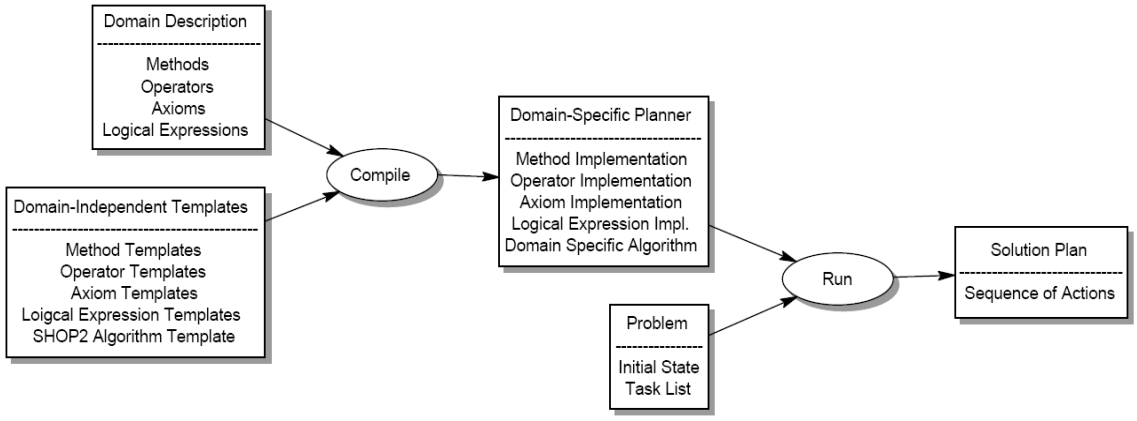

Figure 4.7: JSHOP2 Compilation Process [27]

Figure 4.7 depicts the compilation process of JSHOP2. From the figure it is clear that the generation of the solution plan involves two phases, each consisting of two different sets of inputs. The first set consisting of domain description and domain-independent templates is fed into JSHOP2 for compilation. The second set consists of domain-specific planner (that is the output from the compilation phase), and the problem description.

In the first phase, a domain description is fed into JSHOP2. The domain description is composed of operators, methods, axioms and logical expression. Both the operators and methods include logical expressions that describe their preconditions. In JSHOP2, an operator takes the following form:

$$(:\ operator\ \ h\ \ pre\ \ del\ \ add\ \ [c])$$

where

- h is the operator's head – an operator begins with an exclamation mark and denotes a primitive task;
- *pre* is the precondition in logical expression form;
- *del* is a delete list and its elements could be any of the following:
  - a logical atom
  - a protection condition

– an expression

- *add* is an add list of logical atoms that has the same form as *del*;
- $[c]$ is the operator's cost. If omitted, the cost is 1.

A method is written in the following form:

$$(:\mathit{method}\ \ h\ \ [n_1]\ \ C_1\ \ T_1\ \ [n_2]\ \ C_2\ \ T_2\ \ \ [n_k]\ \ C_k\ \ T_k)$$

where

- h is the method's head (which is a task atom);
- C_i is the precondition in a logical expression form;
- T_i is the method's tail (which is a task list);
- n_i is the name for $C_i\ T_i$ pair. These names can be omitted and a unique name will be assigned for each pair.

The following example (Listing 4.4) considers an online bookstore.

```
;;;
;;; Declaring the data
;;;
    (:operator (!purchase ?a) () ()
               ((have ?a)))
    (:operator (!transferFunds ?amount) () ()
               ((fundstransferred ?amount)) )
    (:operator (!purchase ?amount) () ()
               ((fundstransferred ?amount)) )
;;;
;;;The methods for the tasks
;;;
    (:method (buy ?author ?title ?amount)
        ((shoplist (item ?item) (author ?author)
                (title ?title) (qty ?quantity)
                (price ?price))
                (call >= ?amount ?price))
        ((!purchase ?author) (!noproblem ?author)
         (!transferFunds (call - ?y ?price)))
        )
```

Listing 4.4: An online bookstore domain

The example shows three primitive operations that are responsible for checking the availability of the book, checking for sufficient funds and initiating payment.

`(:operator (!purchase ?a) () () ((have ?a)))` checks the availability of the `book(a)`. The precondition and the delete list were left empty, because there is no precondition that needs to be satisfied at this stage and nothing needs to be removed from the list. Furthermore, `(:operator (!transferFunds ?amount) () () ((fundstransferred ?amount)) )` is the basic operation to transfer the funds from the user. Lastly, `(:operator (!purchase ?amount) () () ((fundstransferred ?amount)) )` is used to confirm the purchase by checking whether the amount required to purchase the book has been successfully transferred into the account or not.

The method that follows from the operations is responsible for the actual purchasing process. It first takes in a list that is the description of the book the user requested and checks for availability. The availability of the book is the first precondition that needs to be satisfied. Secondly, it checks for sufficient funds in the user's account using `(call >= ?amount ?price)`. This is the second precondition that needs to be satisfied. If both preconditions are satisfied, then it will process the transfer and complete the purchase.

The axioms are 'horn clause'-like statements that can be used to infer preconditions that are not explicitly presented in the current state of the world. An axiom has the following form:

$$(:- \ a \ [name_1] \ E_1 \ [name_2] \ E_2 \quad [name_n] \ E_n)$$

where

- a is the axiom's head, and
- $[name_i]$ is the name of the expression E_i.

Besides the domain description, another set of input to the JSHOP2 is the domain-independent templates for methods, operators, axioms, logical expressions and SHOP2 algorithms. For each corresponding element occurring in the domain description, the template will be instantiated accordingly.

The result after the compilation is a piece of code that implements the behaviour of the corresponding element and a specific instance of SHOP2 algorithm for the domain.

To solve the action problem in a compiled domain, one needs to know the domain of the problem before it could generate any plans as output. Therefore, a problem description has to be compiled. The following syntax is used to define the domain:

$$(defdomain\ \ domain-name\ \ (i_1\ \ i_2\quad i_n)$$

where

- $domain-name$ is a symbol which represents the problem domain, and
- i_i could either be an operator, a method or an axiom.

The problem description consists of an initial state and a task list and it has the following form:

$$(defproblem\ \ problem-name\ \ domain-name\ \ (a_1\ \ a_2\quad a_n)\ \ T)$$

where

- $problem-name$ and $domain-name$ are symbols that represent the problem and the problem domain respectively;
- a_i is a logical atom, and
- T is a task list.

The following code (Listing 5.1) illustrates an online shopping problem domain (ref. Listing 4.4).

```
(defproblem problem Webservice
    ((shoplist (item Textbook) (author Bishop)
               (title Java) (qty 1) (price 65))
    )
    ((buy Bishop Java 70))
)
```

Listing 4.5: An online bookstore problem domain

In this problem domain, we specify the book we would like to purchase as a shoplist, and the task that needs to be performed is the purchasing of the book specified. `(buy Bishop Java 70)` denotes that the book we would like to purchase is by Bishop called Java and we have $70.

A problem template is then again instantiated according to the problem description. The result is an execution code (in Java) which can be run to solve the problem. The solution to the problem is represented as a sequence of actions. An example of the plan generated from the online bookstore is shown in 4.8.

```
Plan #2:
Plan cost: 2.0

(!purchase bishop)
(!noproblem bishop)
--------------------

Time Used = 0.0
```

Figure 4.8: Solution to the online bookstore problem in JSHOP2

4.2.3.2 Evaluation

SHOP2/JSHOP2 does not follow the approach of the SIPE-2 and O-Plan, where monitors are used during run-time to monitor the execution of the plan and to react to unexpected events as they may arise. Instead, JSHOP2 generates a plan based on the current world state and executes the plan later. It guarantees that a plan will be found if all the necessary requirements are satisfied at the time of plan generation. As such, JSHOP2 lacks the built-in notions of unexpected events. That is, we have to assume we know everything about the state of the world, and know exactly how our actions will affect the state. Therefore, there will never be a situation where the state of the world is not what was expected. However, the effort required to create a knowledge base of domain-independent information can be tedious.

Some advantages of SHOP2/JSHOP2 include ordered task decomposition, the combination of HTN decomposition and the reasoning power in the

preconditions. It provides the ability to write domain-dependent knowledge bases that contribute to efficient planning performance. The ability to plan in the order that those tasks will be performed makes it possible to know the current state of the world at each step in its planning process. In this way, it is possible to incorporate significant reasoning power in the precondition evaluation. While HTN decomposition focuses the search on the goal, the reasoning over the precondition prunes the search space by eliminating inapplicable methods and operators in the search spaces.

4.3 Summary of Evaluation

The preceding sections evaluate each individual HTN planning system with respect to the list of self-healing requirements identified in Section 3.4. Table 4.5 summarises the findings that emerged from the evaluation.

Table 4.5: Evaluation summary of HTN planner

Planning system	**Monitoring**	**Analysing**	**Sense-making**	**Replanning**	**Executing**
SIPE-2	Execution monitor	Problem recogniser	Problem recogniser	General re-planner Replanning actions	Planner Execution monitor
O-Plan	Execution monitor	Knowledge sources	Knowledge sources	Replanning algorithms	Plan generation Execution monitor
JSHOP2	Assume to have knowledge about the world at any given time				

The features offered by SIPE-2 and O-Plan satisfy the self-healing requirements identified. On the other hand, JSHOP2 takes a different approach as opposed to SIPE-2 and O-Plan. In particular, the plan representation of JSHOP2 is an ordered network, meaning that the planner plans for tasks in the same order that they will later be executed. Using this approach greatly reduces the complexity of the planning process by avoiding some task-interaction issues.

Another feature that distinguishes JSHOP2 from the other two HTN planning systems is the way it reacts to unexpected events and failures. Technically speaking, JSHOP2 does not have any run-time monitoring techniques available. However, when specifying the domain description, the developers can put in various methods that can be adopted during plan generation. In the case where the method does not generate a desirable plan, an alternative method is selected, and so forth.

That means the domain should have sufficient knowledge of the world state to anticipate any changes in order to write methods to handle unexpected events or failures. Having to anticipate all the world states is a tedious procedure and nearly impossible to do in a real-life domain. If the planner fails to find any methods that would generate a satisfactory plan, it will return a 'zero plan found'.

In terms of monitoring, SIPE-2 and O-Plan both have a run-time monitor available. While SIPE-2's execution monitor constantly receives information about the plan that is being executed and about unexpected events as they arise, it does not monitor the world directly. Instead, upon encountering unexpected events, it translates their description into its own language before replanning is initiated. Information on how the translation is performed is not clear. By performing the translation, the monitor could potentially hamper the performance of the replanning. However, considering the fact that SIPE-2 supports resource constraints, the translation process is not going affect the overall performance. This is because the current available resources will be checked periodically against the resources required for the translation. In O-Plan, sensors and agents are used for monitoring the plan and reporting unexpected events and/or failures to the execution monitor. Descriptions of these failures are not available.

According to the self-healing cycle (ref. Figure 3.7), the run-time monitor should first try to re-execute the plan once it has detected a problem. Replanning should be performed only if re-execution has failed. The Web service that it is trying to access could probably be slow, and re-executing the plan a few times could potentially resolve the problem. However, none of the HTN planners has the ability to automatically re-execute the plan before

performing replanning. In the Web service paradigm, this can cause lots of unnecessary replanning. SIPE-2 and O-Plan both offer mixed-intuitive planning, which means the user can interact with the planning system during planning. In this way, users would manually initiate the re-execution of the plan. To re-execute the plan in JSHOP2, we have to force the planner to call the same method repeatedly.

During replanning, the nature of the problem needs to be analysed and suitable replanning actions must be applied. In SIPE-2, the problem recogniser is responsible for both analysing and reasoning (i.e. Sensemaking) about the problems. Given the representation of the SIPE-2 plan, there are only six problems that could occur, hence there is a predefined finite set of problems. The replanning part of SIPE-2 tries to change the old plan, using the heuristics to retain significant parts of its plan wherever possible. An important contribution of SIPE-2 is its general set of replanning actions that are used to modify plans. These replanning actions are used in the replanner and have the potential to facilitate the addition of domain-specific knowledge about error recovery, as the user could specify which replanning actions to take in response to certain anticipated errors. After the problems have been successfully identified, the general replanner applies suitable replanning actions to the problem and the end result is a plan with an unresolved goal. This plan is then passed to the planner to perform the necessary planning in order to come up with a solution. In many cases, SIPE-2 is able to retain most of the original plan by making some modifications, and also capable of shortening the original plan when serendipitous events occur.

O-Plan offers two distinctive plan repair algorithms that deal with: 1) execution failure, and 2) unexpected world events. A number of knowledge sources are used when dealing with plan repairs. The knowledge sources are responsible for reasoning about the problems detected and deciding what actions need to be taken to repair the affected plan. Plan repair is done through backtracking. During replanning O-Plan tries successive combinations of schemas and variables, without any indication of whether these combinations are likely to be better or worse than the ones it has tried before. By trying various combinations in such a way, O-Plan may perhaps exhaust

its resources before finding an optimal plan.

4.4 Conclusion

In this chapter three HTN planning systems - SIPE-2, O-Plan and JSHOP2 - were chosen for detailed evaluation against the self-healing requirements identified in Chapter 3. The evaluation outcomes revealed the capability of HTN planning systems in self-healing Web service composition at a theoretical level. In particular, reactive strategy planning, SIPE-2 and O-Plan, seems more promising than JSHOP2. What remains, is to demonstrate the practical applications of the HTN planning systems in self-healing Web service composition processes. Out of the three planning systems discussed in this chapter, the author was unable to make use of the SIPE-2 due to licensing issues. In the next chapter experiments using O-Plan and JSHOP2 are conducted to address their practical applications in self-healing Web service composition.

Chapter 5

Experiments

5.1 Introduction

At this point the reader should be conversant with the self-healing Web service composition problem. Chapter 2 provided an overview of the composition process and its relevant technologies. Next, Chapter 3 clarified the problems in Web service composition and identified five requirements needed for self-healing Web service composition. Lastly, Chapter 4 examined the available HTN planners, and to this end the features of each planner have been mapped out to the requirements identified in Chapter 3. What remains now is to check the applicability of HTN planners in real application with respect to the faults and requirements mentioned in Chapters 3 and 4.

Chapter 5 reports on the experiments conducted on the applicability of the HTN planners in self-healing Web service composition. The experiments were conducted using the O-Plan and JSHOP2 planners. The goal of the experiments was to test the proof-of-concept as was identified in the preceding chapters.

5.2 Experimental Setup

Two planners were chosen for our experiments, namely O-Plan and JSHOP2. Our choice was based on the evaluation conducted. O-Plan was found to satisfy all the requirements of the self-healing cycle, while JSHOP2 has to make

assumptions about the current state of the world. Thus it was considered worthwhile to investigate whether the concept of plan first and execute later has any advantages over a reactive strategy planning.

The experiments are of a proof-of-concept nature. Their focus is not on evaluating the computational properties *per se*. In fact, the experimental studies are intended to obtain basic insights into the following questions:

- How does an HTN planning system match the requirements of self-healing Web service composition?
- What faults can be handled by the HTN planning system?
- Can an HTN planning system handle unknown faults?

Unknown faults are defined as faults that cannot be identified as one of the groups in our taxonomy.

To illustrate how O-Plan and JSHOP2 fare in answering the above questions, the online shopping domain described in Section 2.4 will be used for the experiments.

For the purpose of replanning, we expanded the shopping domain to include two dummy web catalogues and a dummy credit checker as rebinding services. Figure 5.1 depicts a detailed operation taking place in the shopping domain. The Initial State is responsible for initiating the composition. It sends through a shopping service request in the form of `SR=(ItemList)`, which defines one type of problem in the shopping domain. For the sake of simplicity, our experiments will look only at the online bookstore problem. An example of a shopping list (shopping for two items) is as follows:

```
((shoplist (
    ((item Textbook) ((author Bishop)
     (title "Java Gently")) (qty 1))
    ((item Fictionbook) ((author King)
     (title "The Eyes of the Dragon")) (qty 1))
)))
```

Listing 5.1: An example shopping list for the online bookstore problem

Each Web catalogue consists of a set of items and their features. For each item, information on its features, prices and current availability is presented. The current availability of an item is also updated during execution.

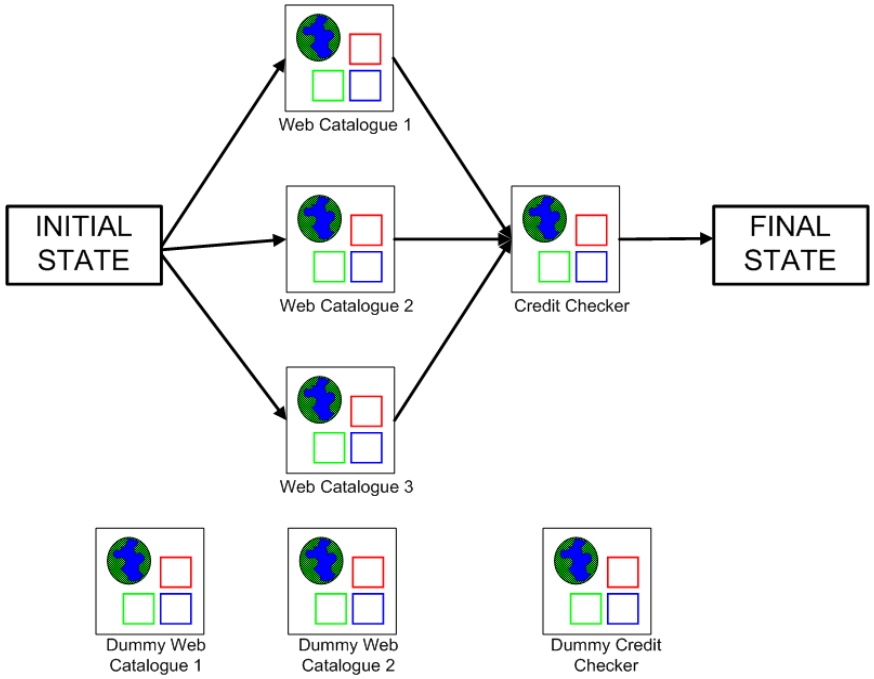

Figure 5.1: An expanded shopping domain

After a request has been sent through, the planning system tries to establish a plan that will satisfy the requirements. This step involves searching through the current participating Web catalogues that have the items specified and checking on their availability. Once such an item has been found and all the requirements have been met, it will go through to the checkout process. The credit checker checks for sufficient funds before payment is processed, and a plan is generated. During the execution of a plan, various verification and monitoring measures have to be taken into account. For example, if the current participating Web catalogues fail to satisfy the requirement, a recovery mechanism is needed to compromise the situation. The dummy web catalogues and dummy credit checker are there to act as replacements in the event that the current participating Web services fail to deliver their service as expected.

In Table 5.1, a list of test cases is presented, together with the effects we are trying to achieve when faults (cf. Section 3.3) are triggered (injected manually into the planning system).

Table 5.1: Test cases of an online bookshop domain

Case	Test Case	Effects
1	Get the price of a specific item	- Unresponsive Web service - Slow service - Incorrect result
2	Purchase a specified item	- Unresponsive Web service - Slow service - Incorrect result
3	Find the best prices of a specified item; then purchase	- Unresponsive Web service - Slow service - Incorrect result - Incoherent results

The following three strategies are used for each of the faults triggered:

- Strategy 1: Generation of a plan followed by a single attempt at execution.
- Strategy 2: Generation of a plan, followed by multiple attempts at execution. If failures occur, the original plan is re-executed.
- Strategy 3: Generation of a plan, followed by multiple attempts at execution. If failures occur, replanning is performed, followed by execution of the new plan.

Table 5.2 summarises the failure setting and the strategies to be followed:

Table 5.2: Failure settings and strategies

Strategies	Fault setting	Expected Outcomes
Strategy 1	Without fault	Execute normally
	With fault	Fail
Strategy 2	Without fault	Execute normally
	With fault	Possible to continue the execution if faults were caused by slow and unresponsive service.
Strategy 3	Without fault	Execute normally
	With fault	Execute normally after replanning

5.3 Hypotheses

In view of the goal of our experiments, the following hypotheses were formulated:

Hypothesis 1: HTN planning will be able to find a plan if a plan exists and it will execute this plan successfully.

Hypothesis 2: To re-execute a plan takes less time than to repair the plan. However, this is only possible if the Web service has not failed completely (i.e. due to slow and unresponsive service).

Hypothesis 3: The HTN planning system is able to perform replanning when re-execution of the plan fails. Replanning may also identify alternative shorter plans.

Hypothesis 4: Incoherent results will not occur.

5.4 Results

In this section, the results of our experiments will be discussed. For illustrative purposes, test case 3 was chosen to demonstrate the power of O-Plan and JSHOP2 in self-healing Web service composition. For full details of

other test cases in O-Plan and JSHOP2, please refer to Appendices A and B respectively.

Based on the fault outcomes and their origin (deduced from Table 3.2 (Section 3.3)), the relevant faults to be triggered were selected in order to obtain the desirable effects as shown in Table 5.3. The first column of the table shows the faults to be triggered in the experiments and their corresponding effects are shown in the second column.

The faults related to SLA and QoS will not be tested in the experiments, since they are non-functional and beyond the scope of this study. The aim of the current study was to concentrate only on the functional aspect of the composition.

Table 5.3: Faults and effects

Faults to be triggered	**Effects**
Interface change	- Unresponsive Web service - Incorrect result - Incoherent results
Time-out	- Unresponsive Web service - Incorrect result - Slow service
Workflow inconsistency	- Unresponsive Web service - Incorrect result - Incoherent results

5.4.1 Experimental Results (O-Plan)

Given that the Task Formalism is correctly defined, we would like to generate a plan in O-Plan that satisfies a specific problem. A plan generated in O-Plan does not show the value of each action taken; instead it shows what actions need to be taken in order to satisfy the user's request. For example, if we would like to purchase a book authored by `Bishop` called `Java` for the best price available, then the following actions (refer to Listings 5.2) need to be performed within a task to generate a plan.

```
task buy_book_all_cases;
vars ?author=?{not none}, ?title=?{not none};
  nodes
     1 start,
     2 finish,
     3 action {get_price ?author ?title},
     4 action {buy_book ?author ?title},
     5 action {get_best_price ?author ?title},
     6 action {purchase book};
  orderings 1 ---> 3, 3 ---> 4, 4 ---> 5, 5 ---> 6, 6 ---> 2;
end_task;
```

Listing 5.2: O-Plan code for purchasing a book

The task has local variables that capture the user request and the actions need to be performed in order to generate a plan. Each action is defined as a number of schemas, which are responsible for network expansion. Once all primitive actions have been identified, they are solved individually, and the desired effect is achieved. The output of case 3 in our example is shown in Figure A.5 in Appendix A.

Observation: Given the correct semantics of the request, O-Plan was able to generate a plan that satisfies the requirement. O-Plan goes through a series of steps and eventually ends the plan on NODE-2, at which stage it is executed properly. It uses a backtracking mechanism for plan generation, so if there is such a goal that can be reached, then the plan will find it.

Various faults were triggered to test O-Plan's response. In O-Plan the specification of a fault is difficult, and often there is no clear distinction between different faults. Therefore, we randomly removed the available services in order to trigger faults such as interface change and workflow inconsistency. When we used strategy 2 (re-execution), we were unable to recover any interface change and workflow inconsistency faults. And it terminiates the execution at the end node (NODE-4).

Each task in O-Plan may be given a specific time frame for execution. In the event of the task failing to meet the time frame specified, that particular task is skipped and the task that follows is commenced with. However, this implies that a failure has occurred. Re-execution of the plan may fix the problem, depending on execution of the task. If re-execution is successful,

the normal path is followed and the process ends on NODE-2. If not, it will end on NODE-4.

Observation: When a time-out fault is triggered, it is possible that the original plan will be selected once again after re-execution. However, when interface change and workflow inconsistency faults are triggered, re-execution of the plan cannot fix the problem.

From our observation we noticed that strategy 2 (re-execute) was able only occasionally to fix the time-out faults and it was not able to fix any other faults that were triggered. Therefore, strategy 3 was next applied to test O-Plan's ability in replanning. Figure A.6 (in Appendix A) illustrates a plan generated after replanning has been applied to the problem. Various contingency plans were provided to handle faults as they may arise. In the event of the book with a specific price being out of stock (see code below), then either NODE-4 or NODE-8 will be reached, depending on the input. For example, the lowest price of the book `Java` by `Bishop` in the database is $35. However, the quantity counter of the book that is sold at this price indicates that it is out of stock. Therefore, the operation will be terminated and alternatives will be looked for.

```
resource_types consumable_strictly {resource book
(item Textbook) (author Bishop) (title Java) (qty 0) (price 35)};
```

The replanning procedure goes through each node, starting from NODE-9, moving up to NODE-6 and then NODE-5 if necessary, to look for a plan that can satisfy the user request. If such a plan exists, then it will execute normally and the process will end on NODE-2. NODE-3 ACTION UNAVAILABLE TITLE and NODE-7 ACTION UNAVAILABLE AUTHOR are flags used to determine availability after a plan has been generated of that specified book. In our test cases, the quantity counter of the book we purchased indicated zero, and therefore created a flag to report that the book was no longer available.

Observation: Replanning was able to fix all the faults that had been triggered by choosing alternative plans. Despite the incomplete information to the planner, it was able to reason upon the given input and performed

replanning based on this information. However, if none of the input can be reasoned upon, O-Plan will simply return an error message indicating that it is not possible find a plan, for example:

```
Plan option n - Planner out of alternatives
```

where `n` is the number of the task that you chose to run.

5.4.2 Experimental Results (JSHOP2)

Given that the problem domain and the problem description are correctly defined, we would like to know whether JSHOP2 is capable of generating a plan to solve a specific problem. For example, we would like to pay the best price for the book Java authored by Bishop and the maximum amount that we are willing to pay is $100. In this case the request will look as follows: `(getBestPrice db Bishop Java 100)`. During the experiment, we sent through multiple requests and the results showed that the planning system is indeed able to generate a plan and execute successfully if a plan exists. An example of the request mentioned above and the corresponding output is shown in Listing 5.3:

```
1 plan(s) were found:
2
Plan #1:
Plan cost: 2.0

(!displayprice bishop java 20.0)
7 (!purchase bishop java 20.0)
----------

Time Used = 0.016
```

Listing 5.3: Plan for case 3 in JSHOP2

During the plan generation process, JSHOP2 goes through a list of methods and makes use of the methods that are appropriate to the current problem. When a method is invoked, the preconditions and constraints are checked. Thus, it provides verification of the plan that is being generated.

Observation: When the request and the methods that cater for the world condition are correctly defined, JSHOP2 will be able to generate a

plan that satisfies the requirement. Various faults are triggered to test the resilience of the composition. The first fault that we looked at was the fault triggered by interface change. This fault can result in unresponsive Web service, as well as incorrect and incoherent results.

```
(getBestPrice db Bishop Java 100)
(getBestPrice db Bishop)
```

The request was changed so that it would contain only the name of the author. The results, using Strategy 1, showed that the planning system was not able to generate a plan to reach a desirable output. Our request was then executed multiple times, and the same results were generated. After several consecutive failed attempts to generate any plans, strategy 3 was used. To incorporate replanning, multiple methods were introduced with various parameters that cater for all possible situations. The results after replanning showed all the books that were authored by Bishop. It was discovered that if there were no alternative methods to cater for any interface change, JSHOP2 was not able to generate any plan.

Observation: When an interface is changed, it causes the service to become unavailable (i.e. unresponsive). Since the service cannot be called upon, JSHOP2 returns the result 'no plan found'. Of course, one cannot conclude that it returns an incorrect result, yet to return no result is also not desirable. Therefore, from the user's point of view, it indeed provides an incorrect result. Lastly, regardless of the number of re-executions, the results do not vary during execution. Henceforth it seems that incoherent results will not occur when a plan is generated using JSHOP2.

The next fault triggered was the time-out fault. As was expected, JSHOP2 was unable to generate a plan (strategy 1), even after several re-execution attempts (strategy 2). The replanning strategy was then implemented. To facilitate such replanning mechanism, the replanning method was incorporated in the domain description. The result after replanning was that we found the best price for the book `Java` by `Bishop` and the purchase process was completed. `((!replan ?author) (reassignBestPrice ?s ?newPrice ?author ?title))` was responsible for initiating the replanning process. It called

upon the alternative `reassignBestPrice` method to perform the operation of finding the best price.

```
(:method (reassignBestPrice ?s ?l ?author ?title)
        ((shoplist (item ?item) (author ?author) (title ?title)
                   (qty ?quantity) (price ?price))
            (lowest (lowestPrice ?lowest))
            (call <= ?price ?lowest)
            (assign ?lowest ?price)
            )
        ( (!displayPrice ?author ?title ?price)
          (!purchase ?author ?title ?price) )
)

    (:method (getBestPriceUnresponsive ?db ?author ?title ?newPrice)
    (forall (?s)
       (shoplist ?s)
        ((in ?s ?db)
        ((call <= ?newPrice ?price)
        (assign ?newPrice ?price)
        )
        )
    )
    (
        ((!replan ?author) (reassignBestPrice ?s ?newPrice ?author ?title))
    )
    )
```

Listing 5.4: Re-planning method for time-out fault

Observation: A time-out fault not only causes the service to become unavailable, but also slows down the service. Since the service cannot be reached, the result returned by JSHOP2 is 'no plan found'. Thus, for the same reason that an interface change causes the incorrect result, the time-out fault also produces an incorrect result.

Lastly, the workflow inconsistency fault was triggered. It was observed that the first two strategies did not work in response to workflow inconsistency. The replanning mechanism was able to pick up the fact that the book title was misspelt and returned all the books by this author. Some would have expected the planner to return results that contained only the Java books. However, based on the knowledge presented, the planner only acknowledged that the book title does not exist, but recognised the author. As a result, it returned all the books written by the author concerned.

5.4.3 Summary of Experimental Results

In Section 5.3, a number of hypotheses were proposed regarding the ability of O-Plan and JSHOP2 in self-healing Web service composition. These expectations will be discussed next in the light of the presented results. Referring to the hypotheses that were established at the beginning of these experiments, the following can be noted:

Hypothesis 1: [*HTN planning will be able to find a plan if a plan exists and it will execute this plan successfully.*] This was confirmed in all cases.

Hypothesis 2: [*Re-execute a plan takes less time than to repair the plan. However, this is only possible if the Web service has not failed completely (i.e. due to slow and unresponsive service).*] Unless it changed the method in the domain description, JSHOP2 was unable to resolve the problem that had been caused by the unresponsive service. O-Plan, on the other hand, occasionally succeeded in fixing the problem, depending on each individual execution.

Hypothesis 3: [*HTN planning system is able to perform replanning when re-execution of the plan fails. Replanning may also identify alternative shorter plans.*] During replanning, O-Plan generated various alternative plans. However, the plans generated were not shorter than the original ones.
Replanning in JSHOP2 did not return a shorter alternative plan. Instead, in most case, it returned more alternative plans with various combinations. For a more detailed description, see the results shown in Appendix B: Table B.1 (Interface changed and incorrect service faults).

Hypothesis 4: [*Incoherent results will not occur.*] This was confirmed. Each plan generated was executed multiple times throughout the course of the experiments to ensure that the results were coherent.

Based on our observations, answers can now be provided to the questions that were set out at the beginning of this chapter, namely:

Q: How does an HTN planning system fit into the requirements of self-healing Web service composition?

O-Plan fulfils the self-healing requirements for Web service composition, except for the fact that re-execution of the plan has to be triggered manually by the user.

JSHOP2 partially fulfils the self-healing requirements for Web service composition, namely planning and verification, and execution without modification of the domain specification. The other requirements execution monitor, analyser, sensemaking and replanning are implemented as methods in the domain specification by the developer. Lastly, re-execution is done manually by the user.

Q: What faults can be handled by the HTN planning system?

O-Plan is able to handle interface change and workflow inconsistency faults. However, the exact details of how the planner describes the faults are unclear. (In our experiments the available Web services were randomly removed in order to trigger the faults.) Furthermore, each task in O-Plan is given a specific execution time-frame. If the task fails to meet a specific time-frame, it will be re-executed and sometimes such faults can be fixed. Alternatively, replanning will generate an alternative path for execution. O-Plan therefore proves to have the ability to handle time-out faults.

JSHOP2 is also able to handle the time-out, interface and workflow inconsistency faults. However, this is done through the use of methods defined in the domain specification.

Q: Can an HTN planning system handle unknown faults?

Since the exact details of how the faults are described in O-Plan are unknown, it is difficult to pinpoint whether it is possible to handle unknown faults. It was observed that if none of the inputs can be reasoned about by the planner, then it is unable to generate a plan. On the other hand, if one of the inputs to the planner can be reasoned about, then it is able to generate a plan. Therefore, it was concluded that O-Plan may be able to handle unknown faults as long as one of the inputs to the planner can be reasoned

about.

JSHOP2 can partially handle unknown faults, provided that at least one of the parameters in the query is valid. The result showed that JSHOP2 was unable to pinpoint the exact problem (the unknown fault), and therefore it returned all the results that contained valid parameters. More details can be found in Appendix B: Listing B.6.

5.5 Conclusion

This chapter presents the practical results of HTN planning systems in self-healing Web service composition. Starting with the experiments, the book-store example shown in Figure 2.5 (Section 2.4) was refined to incorporate a set of alternative services for replanning purposes. A list of faults to be triggered and their effects were deduced from Table 3.2 (Section 3.3), which aided the investigation of how HTN planning systems respond to those faults.

The results of our experiments were presented, and the faults and HTN planning systems' response to those faults were examined according to the model defined in Section 3.4. After summarising the results of these experiments, it was concluded that the HTN planning systems can indeed be used for self-healing Web service composition. Moreover, reactive strategy planning has proven to have advantages over planning that plan first and execute later, as it does not have to make assumption and predict the world at design-time.

This chapter completes the main contents of the dissertation. The next and final chapter will discuss the achievements of the current study and draw some general conclusions from them. Recommendations on possible future work will also be presented in Chapter 6.

Chapter 6

Conclusions and Recommendations

6.1 Summary

In the introduction to this dissertation, it was stated that although the Web service paradigm allows applications to electronically interact with one another over the Internet, the integration of Web services with respect to failure handling is still lacking at this point. Self-healing Web service composition was selected as the topic for discourse.

Throughout the current study, various problems associated with Web service composition were encountered. To better understand each of these problems, a fault taxonomy was proposed. The identification of classes of faults has helped to abstract the problem and organise new faults to automatically identify the most suitable reactions. Three categories of faults were defined and discussed, namely physical faults, development faults and interaction faults. These three categories were further expanded to incorporate three distinctive groups of viewpoints to add knowledge about where the faults originated from. The viewpoints included the phase of occurrences, system boundaries and system dimensions. The fault taxonomy that was presented also gave rise to the issue of fault recovery and continued process execution. From this, a list of self-healing requirements for Web services composition was identified. The requirements identified were used to construct a self-

healing cycle, which in turn demonstrated how a composition should react when an unexpected event occurs during the execution.

To facilitate the use of the self-healing cycle proposed, three HTN planning systems were evaluated based on the requirements defined. Reasons as to why a particular module of the planning system satisfies a particular requirement was also offered. Based on the evaluation outcomes, it was observed that the HTN planning systems were able to perform self-healing Web service composition.

The theoretical evaluation of the HTN planning systems was substantiated through practical experiments. The purpose of the experiments was to investigate the impact of HTN planning systems in self-healing Web service composition. A series of tests was conducted to gauge how the HTN planning systems withstand both normal and abnormal behaviours of the composition process. The test results suggested that HTN planning systems were able to meet the self-healing requirements. The results also suggested that they have a significant impact on retaining the correctness of the composition even during unexpected occurrences.

6.2 Limitations

Throughout this dissertation, several limitations have been identified and they are:

- **Only a single domain has been investigated.** The test scenario presented in this dissertation, namely a shopping domain, was limited to sending a shopping list consisting only of books. Only three test cases were use for the experiments. Such limitation was mainly due to the manual injection of faults into the planning system.

- **A limited set of faults have been triggered.** Only three faults were triggered in the experiments and they were not formally specified. Faults should be expressed in a specification language, and the development of a fault simulator can then be used to inject faults into the planning systems for testing purposes. In such a way, even the small-

est defects which may remain hidden due to manually injection can be ironed out, especially the faults that are related to QoS and SLA (non-functional aspects of a composition). However, this was considered beyond the scope of this dissertation.

- **Only HTN planning systems have been studied.** Studies of other planning systems and their applications in self-healing Web service composition should be carried out in order to compare and identify their strengths and weaknesses.

6.3 Recommendations

With regard to Web service composition, contributions have been made in this dissertation by introducing a novel fault taxonomy, as well as defining various requirements that are important in the self-healing composition cycle. The concept of applying HTN planning systems in a self-healing composition context was also presented. Throughout this study several new directions for future research presented themselves. These ideas are summarised briefly below.

Formal specification of fault taxonomy The fault taxonomy for the composition of services, introduced in Section 3.3, identifies possible causes and effects for each of the faults. The purpose of the taxonomy was to determine under which categories of fault detected faults would fall. In order to perform such a cross-reference check, a clear connection between the faults appearing in the taxonomy and a detected fault is needed. Therefore, possible future research could be conducted to formally specify the fault taxonomy in a representation language, so that it could fulfil its purpose.

An evaluation framework The self-healing requirements presented in Section 3.4 were intended for the evaluation of tools/systems in self-healing Web service composition. At the same time, they would also serve as a guideline for the development of self-healing Web service composition applications. Selecting a tool/system that satisfies these requirements can be

problematic, since the existing work on automatic failure recognition and recovery is based on different techniques and they tend to address only a certain type of faults. Therefore, possible future research could focus on the development of an evaluation framework that can analyse the capability of a tool/system and realise the possible improvements that would meet the requirements of the self-healing cycle.

Robust service composition methodology Further investigation into fault behaviour patterns can be used to understand where and when faults will normally occur. This investigation will reveal a behaviour pattern of faults, which will in future lead to the question of how monitoring and recovery mechanisms can co-exist with the faults. Such research work could culminate in a methodology for robust Web service composition.

Appendix A

Shopping Domain in O-Plan

This appendix describes the bookstore shopping domain, implemented in O-Plan version 3.3[7], that was used in Chapter 5.

A.1 Task Formalism

The **bookstore** domain is defined as Task Formalism (TF) in O-Plan, and the code is given below:

```
resource_units dollars = count;
types item = (Textbook Fictionbook);
types author = (Bishop King);
types title = (CSharp Java DesignPatterns TheEyesOfTheDragon);
resource_units qty = count;
resource_units price = count;
resource_units lowest = count;
resource_types consumable_strictly {resource book (item Textbook) (author Bishop) (title CSharp)
                                                  (qty 1) (price 80)};
resource_types consumable_strictly {resource book (item Textbook) (author Bishop) (title CSharp)
                                                  (qty 1) (price 45)};
resource_types consumable_strictly {resource book (item Textbook) (author Bishop) (title Java)
                                                  (qty 1) (price 65)};
resource_types consumable_strictly {resource book (item Textbook) (author Bishop) (title Java)
                                                  (qty 0) (price 35)};
resource_types consumable_strictly {resource book (item Textbook) (author Bishop) (title Java)
                                                  (qty 2) (price 75)};
resource_types consumable_strictly {resource book (item Textbook) (author Bishop)
                                                  (title DesignPatterns) (qty 3) (price 20)};
resource_types consumable_strictly {resource book (item Fictionbook) (author King)
```

[7]Available on `http://www.aiai.ed.ac.uk/~oplan`

```
                                                    (title TheEyesoftheDragon) (qty 1) (price 50)};
resource_types consumable_strictly {resource book (item Fictionbook) (author King)
                                                    (title TheEyesoftheDragon) (qty 1) (price 60)};


resource_types
  consumable_strictly {resource money} = dollars;

task buy_book_all_cases;
vars ?author=?{not none}, ?title=?{not none};
  nodes
     1 start,
     2 finish,
     3 action {get_price ?author ?title},
     4 action {buy_book ?author ?title},
     5 action {get_best_price ?author ?title},
     6 action {purchase book};
  orderings 1 ---> 3, 3 ---> 4, 4 ---> 5, 5 ---> 6, 6 ---> 2;
end_task;

;;;buys a book using the title of the book
task buy_book;
  vars ?item=undef, ?author=undef, ?title=?{not none}, ?qty=undef, ?price=undef, ?bookTitle=undef;
  expands {buy_book ?author ?title};
  only_use_for_effects {location book} = store;
  nodes
     sequential
        1 action {get_price ?title},
        2 action {purchase ?title}
     end_sequential;
  conditions only_use_if {title ?title}={?bookTitle},
             achieve {bought ?title};
  effects {unavailable book (item ?item) (author ?author) (title ?title) (qty ?qty)
                            (price ?price)}, {bought ?author ?title};
end_schema;


schema get_price;
  vars ?item=undef, ?author=undef, ?title=?{not none}, ?qty=undef, ?price=undef, ?bookTitle=undef,
       ?bookAuthor=undef;
  expands {get_price ?author ?title};
  conditions only_use_if {title ?title}={?bookTitle},
     only_use_if {author ?author}={?bookAuthor},
     only_use_if {priceDisplayed ?author ?title};
  effects {book (item ?item) (author ?author) (title ?title) (qty ?qty) (price ?price) exists} = true;
  time_windows duration self = 1 seconds;
end_schema;

schema get_price_author;
  vars ?item=undef, ?author=undef, ?title=?{not none}, ?qty=undef, ?price=undef, ?bookAuthor=undef;
```

```
  expands {get_price ?author};
  conditions only_use_if {author ?author}={?bookauthor};
  effects {book (item ?item) (author ?author) (title ?title) (qty ?qty)
               (price ?price) exists} = true, {priceDisplayed ?author ?title};
  time_windows duration self = 1 seconds;
end_schema;

schema get_price_title;
  vars ?item=undef, ?author=undef, ?title=?{not none}, ?qty=undef, ?price=undef, ?bookTitle=undef;
  expands {get_price ?title};
  conditions only_use_if {title ?title}={?bookTitle};
  effects {book (item ?item) (author ?author) (title ?title) (qty ?qty)
               (price ?price) exists} = true, {priceDisplayed ?author ?title};
  time_windows duration self = 1 seconds;
end_schema;

schema get_best_price;
  vars ?item=undef, ?author=undef, ?title=?{not none}, ?qty=undef, ?price=undef, ?bookTitle=undef,
       ?bookAuthor=undef;
  expands {get_best_price ?author ?title};
  nodes
     sequential
        1 action {get_price ?title ?author},
2 action {buy_book ?title ?author},
        3 action {purchase ?title}
     end_sequential;
  conditions only_use_if {title ?title}={?bookTitle},
     only_use_if {author ?author}={?bookAuthor},
     only_use_if {fn_leq ?price lowest};
  effects {book (item ?item) (author ?author) (title ?title) (qty ?qty) (price ?price) exists} = true,
          {lowest} = ?price, {priceDisplayed ?author ?title};
  time_windows duration self = 1 seconds;
end_schema;

schema get_best_price_author;
  vars ?item=undef, ?author=undef, ?title=?{not none}, ?qty=undef, ?price=undef, ?bookAuthor=undef;
  expands {get_best_price ?author};
  nodes
     sequential
        1 action {get_price ?author},
    2 action {buy_book ?author},
        3 action {purchase ?author}
     end_sequential;
  conditions only_use_if {author ?author}={?bookAuthor},
     only_use_if {fn_leq ?price lowest};
  effects {book (item ?item) (author ?author) (title ?title) (qty ?qty) (price ?price) exists} = true,
          {lowest} = ?price, {priceDisplayed ?author ?title};
  time_windows duration self = 1 seconds;
end_schema;
```

```
schema get_best_price_title;
  vars ?item=undef, ?author=undef, ?title=?{not none}, ?qty=undef, ?price=undef, ?bookTitle=undef;
  expands {get_best_price ?title};
  nodes
     sequential
        1 action {get_price ?title},
    2 action {buy_book ?title},
        3 action {purchase ?title}
     end_sequential;
  conditions only_use_if {title ?title}={?bookTitle},
     only_use_if {fn_leq ?price lowest};
  effects {book (item ?item) (author ?author) (title ?title) (qty ?qty) (price ?price) exists} = true,
           {lowest} = ?price, {priceDisplayed ?author ?title};
  time_windows duration self = 1 seconds;
end_schema;

schema purchase;
  expands {purchase};
  effects {success} = true;
  time_windows duration self =  1 seconds;
end_schema;
```

A.2 Experimental Result Sets (O-Plan)

A.2.1 Case 1: Get price of a specific item

Figure A.1 depicts the output of a plan for obtaining the price of a specific item. When given the correct semantics (i.e. if both the author's name and the book title are supplied), strategy 1 is able to find a plan that finds the price of the book specified. It goes through a series of steps and eventually ends the plan on NODE-2, in which the operation executes properly. However, when a fault occurs whether an interface change fault or workflow inconsistency fault – strategy 2 (re-execute) is not able to recover the fault. Instead it will reach the end node (NODE-4) for the termination of execution. With regard to a time-out fault, each task in O-Plan is given a specific time in which to be executed. In the event of the task failing to meet the time specified, that particular task will be skipped and the plan will go on with the task that follows. A failure therefore occurs. The re-execution of the plan may fix the problem, depending on the execution of that task at run-time.

If the re-execution is successful, then the normal path will be followed and the plan will end on NODE-2. Otherwise it will end in NODE-4.

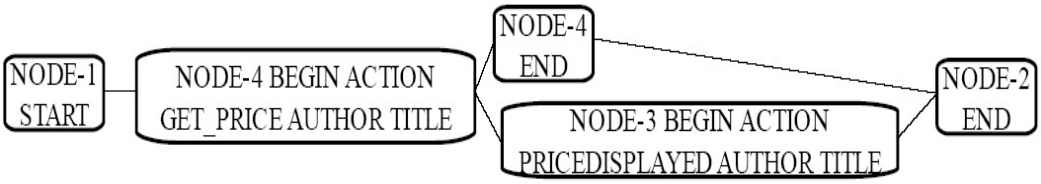

Figure A.1: O-Plan output for Case 1 (Strategy 1 and 2)

Strategy 3 (replanning) for Case 1 (getting the price for a specific item) is illustrated in Figure A.2. Various contingency plans are provided in order to handle faults as they may arise. In case the price cannot be retrieved because the interface has been changed and/or due to workflow inconsistency, then either NODE-4 END, NODE-5 or NODE-6 will be reached, depending on the input. The replanning procedure will go through each node, starting from NODE-6 and moving up to NODE-5 and NODE-4 if necessary, to look for a plan that can satisfy the user request. If such a plan exists, it will execute normally and end on NODE-2.

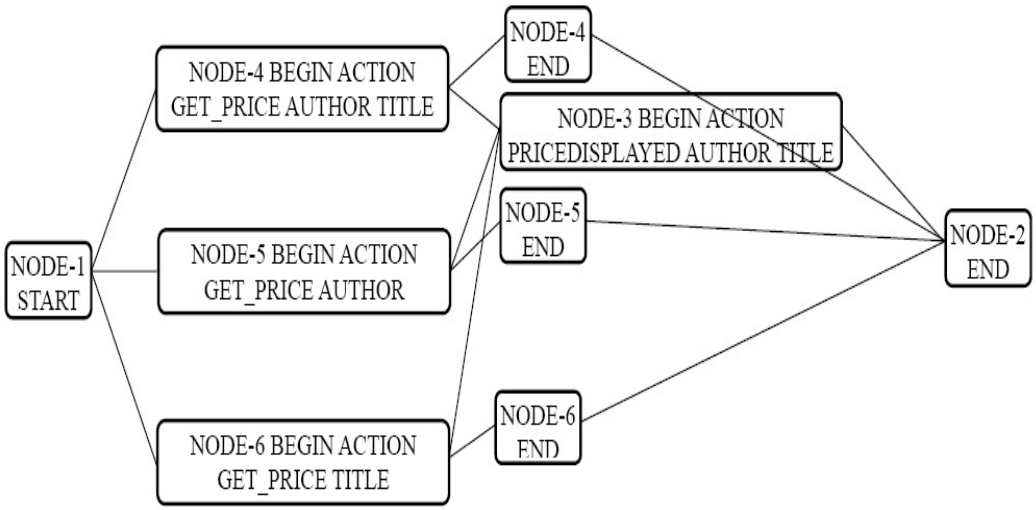

Figure A.2: O-Plan output for Case 1 (Strategy 3)

A.2.2 Case 2: Purchase an item specified

The output of Case 2 is very similar to that of Case 1, except for the series of nodes that need to be followed to purchase a book. The node before NODE-2 acts as a flag to check the availability of a specific book after the purchase has been completed. In our experiment, the quantity counter of the book indicates zero and therefore flags that the book is no longer available.

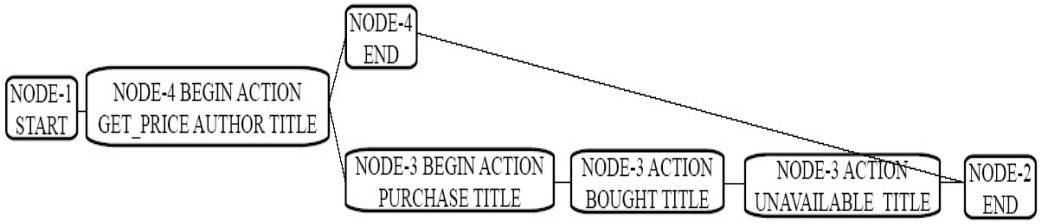

Figure A.3: O-Plan output for Case 2 (Strategy 1 and 2)

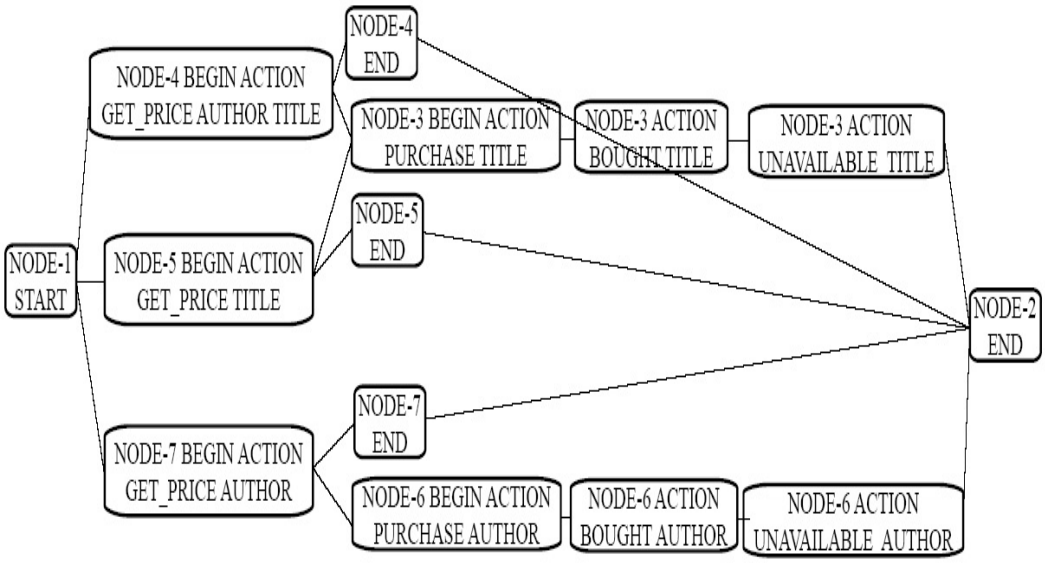

Figure A.4: O-Plan output for Case 2 (Strategy 3)

A.2.3 Case 3: Find the best prices and purchases the item

Case 3 follows the same patterns as Case 2, except that now we also check for the best price of a specific book before purchase.

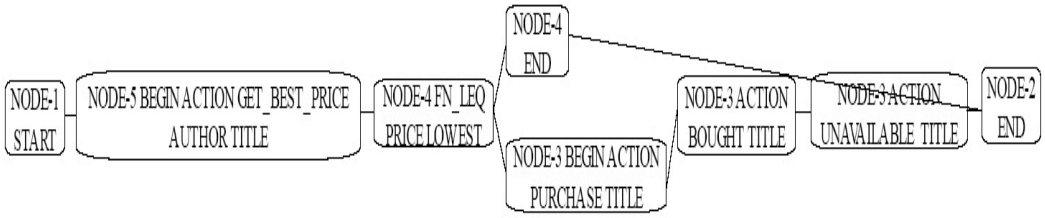

Figure A.5: O-Plan output for Case 3 (Strategy 1 and 2)

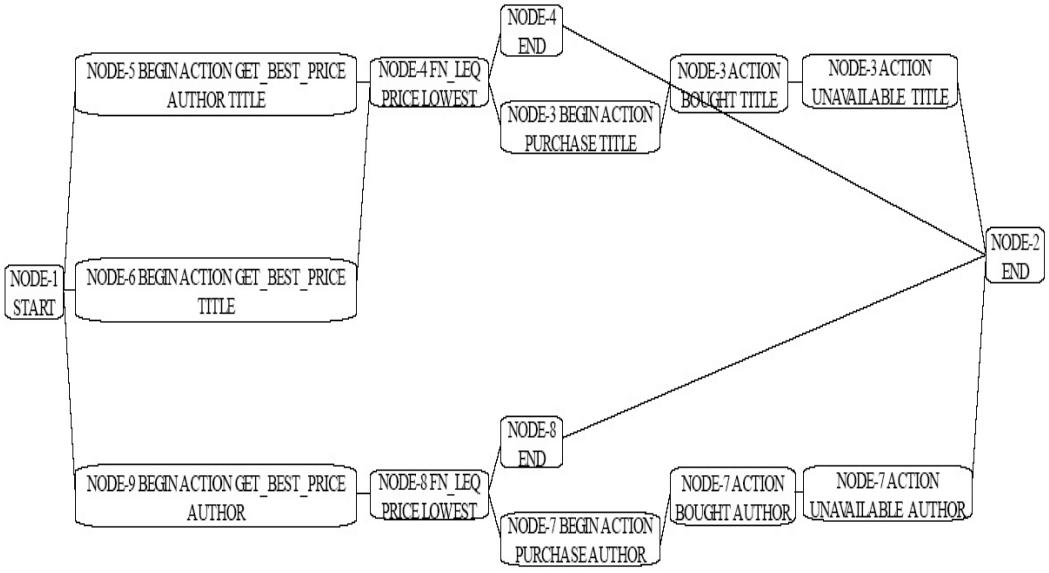

Figure A.6: O-Plan output for Case 3 (Strategy 3)

Appendix B

Shopping Domain in JSHOP2

This appendix describes the bookstore shopping domain, implemented in JSHOP2 version 1.0.2[8], that was used in Chapter 5.

B.1 Domain Definition (JSHOP2)

The **bookstore** domain is defined as follows:

```
(defdomain bookstore
    (

;;;
;;; Declaring the data
;;;
    (:operator (!return ?a) ((have ?a)) ((have ?a)) ())
    (:operator (!nostockerror ?a) (not (have ?a)) ((missing ?a)) ())
    (:operator (!nofunds ?a) (not (have ?a)) ((fundserror ?a)) ())
    (:operator (!noproblem ?a) () () ((have ?a)))
    (:operator (!transferFunds ?amount) () () ((fundstransferred ?amount)) )
    (:operator (!balance ?amount) ()()())
    (:operator (!purchase ?author ?title ?amount) () () ((fundstransferred ?amount)) )
    (:operator (!notrespond ?author) () () ((unresponsive ?author)) )
    (:operator (!incorrectAuthor ?author) () () ((incor ?author)))

    (:operator (!incorrectTitle ?title) () () ((incor ?title)))
    (:operator (!incorrectAuthor ?author) () () ((incor ?author)))
    (:operator (!incorrectPrice ?price) () () ((incor ?price)))

    (:operator (!displayPrice ?author ?title ?price) () () ((success ?author)) )
    (:operator (!noResponse ?author) () () ((slow ?author)) )
```

[8]Available on sourceforge `https://sourceforge.net/projects/shop`

```
    (:operator (!replan ?author) () () ((replan ?author)) )

    (:operator (!bestprice ?author ?title ?price) () () ((foundBestPrice ?author ?title ?price)))


;;;
;;;The methods for the tasks
;;;

   ;;;;;;;;;;;
   ;;;CASE 1;;
   ;;;;;;;;;;;
   (:method (getPrice ?author ?title)
   (shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
   ((!displayPrice ?author ?title ?price) (!noproblem ?author)))

   (:method (getPrice ?title ?author) ;swap the title and author order - interface change
   (shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
   ((!displayPrice ?author ?title ?price) (!noproblem ?author)))


   (:method (getPrice ?author) ;interface change
   (shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
   ((!displayPrice ?author ?title ?price) (!noproblem ?title)))

   (:method (getPrice ?title) ;interface change
   (shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
   ((!displayPrice ?author ?title ?price) (!noproblem ?title)))


   (:method (getPrice ?author ?title) ;incorrect title
   (shoplist (item ?item) (author ?author) (title ?title1) (qty ?quantity) (price ?price))
   ((!displayPrice ?author ?title1 ?price) (!incorrectTitle ?title)))

   (:method (getPrice ?author ?title) ;incorrect author
   (shoplist (item ?item) (author ?author1) (title ?title) (qty ?quantity) (price ?price))
   ((!displayPrice ?author1 ?title ?price) (!incorrectAuthor ?author)))


   ;;;;;method with replanning (when timeout occurs)
   (:method (getPriceUnresponsive ?author ?title)
   (shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
   ((!noresponse ?author) (!replan ?author) (!displayPrice ?author ?title ?price))
   )

   ;;;;;method without replanning (when timeout occurs)
   (:method (getPriceUnresponsive ?author ?title)
   (shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
   ((!noresponse ?author) (getPriceUnresponsive ?author ?title))
   )
```

```
;;;;;;;;;
;;CASE2;;
;;;;;;;;;
 (:method (buy ?author ?title ?amount) ;CASE 2 - Strategy 1
     ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
         (call >= ?amount ?price)
         )
     ((!purchase ?author ?title ?price) (!transferFunds ?price)
      (!balance(call - ?amount ?price)) (!noproblem ?author))
 )

 (:method (buy ?author ?amount ?title)
     ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
         (call >= ?amount ?price)
         )
     ((!purchase ?author ?title ?amount) (!transferFunds ?price)
      (!balance(call - ?amount ?price)) (!noproblem ?author))
 )

 (:method (buy ?title ?author ?amount)
     ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
         (call >= ?amount ?price)
         )
     ((!purchase ?author ?title ?amount) (!transferFunds ?price)
      (!balance(call - ?amount ?price)) (!noproblem ?author))
 )

 (:method (buy ?title ?amount ?author)
     ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
         (call >= ?amount ?price)
         )
     ((!purchase ?author ?title ?amount) (!transferFunds ?price)
      (!balance(call - ?amount ?price)) (!noproblem ?author))
 )

 (:method (buy ?amount ?title ?author)
     ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
         (call >= ?amount ?price)
         )
     ((!purchase ?author ?title ?amount) (!transferFunds ?price)
      (!balance(call - ?amount ?price)) (!noproblem ?author))
 )

 (:method (buy ?amount ?author ?title)
     ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
         (call >= ?amount ?price)
         )
     ((!purchase ?author ?title ?amount) (!transferFunds ?price)
```

```
     (!balance(call - ?amount ?price)) (!noproblem ?author))
)



(:method (buy ?author)
    ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
    (call = ?price ?price)
        )
    ((!purchase ?author ?title ?price) (!noproblem ?author) (!noproblem ?author))
)

(:method (buy ?title)
    ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
        (call = ?price ?price)
        )
    ((!purchase ?author ?title ?price) (!noproblem ?author) )
)

(:method (buy ?amount)
    ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?amount))
        (call >= ?amount ?price)
        )
    ((!purchase ?author ?title ?price) (!noproblem ?author))
)

(:method (buy ?author ?title)
    ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
        (call = ?price ?price)
        )
    ((!purchase ?author ?title ?price) (!noproblem ?author))
)

(:method (buy ?author ?amount)
    ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
        (call = ?price ?price)
        )
    ((!purchase ?author ?title ?price) (!noproblem ?author))
)

(:method (buy ?title ?author)
    ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
        (call = ?price ?price)
        )
    ((!purchase ?author ?title ?price) (!noproblem ?author))
)

(:method (buy ?title ?amount)
    ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
        (call >= ?amount ?price)
```

```
        )
    ((!purchase ?author ?title ?price) (!noproblem ?author))
)

(:method (buy ?amount ?author)
    ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
        (call >= ?amount ?price)
        )
    ((!purchase ?author ?title ?price) (!noproblem ?author))
)

(:method (buy ?amount ?title)
    ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
        (call >= ?amount ?price)
        )
    ((!purchase ?author ?title ?price) (!noproblem ?author))
)


(:method (buy ?author ?title ?amount) ;CASE 2 - Strategy 3 (author misspelt)
    ((shoplist (item ?item) (author ?author1) (title ?title) (qty ?quantity) (price ?price))
        (call >= ?amount ?price)
        )
    ((!incorrectAuthor ?author) (!purchase ?author1 ?title ?amount)
     (!transferFunds ?price) (!balance(call - ?amount ?price)) (!noproblem ?author1))
)

(:method (buy ?author ?title ?amount) ;CASE 2 - Strategy 3 (title misspelt)
    ((shoplist (item ?item) (author ?author) (title ?title1) (qty ?quantity) (price ?price))
        (call >= ?amount ?price)
        )
    ((!incorrectTitle ?title) (!purchase ?author ?title1 ?amount)
     (!transferFunds ?price) (!balance(call - ?amount ?price)) (!noproblem ?author))
)

(:method (buy ?author ?title ?amount) ;insufficient fund
    ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
        (call < ?amount ?price)
        )
    ((!nofunds ?author) )
)


(:method (buyUnresponsive ?author ?title ?amount) ;;;;;;;;;;;;;;Strategy 2
    ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
        (call >= ?amount ?price)
    )
    ((buyUnresponsive ?author ?title ?amount) )
)
```

```
(:method (buyUnresponsive ?author ?title ?amount) ;;;;;;;;;;;;;;Strategy 3
    ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
        (call >= ?amount ?price)
    )
    (
      (!noresponse ?author) (!replan ?author)
      (buy ?author ?title ?amount)
      ;represents slow service.
    )
)


;;;;;;;;;
;;CASE3;;
;;;;;;;;;
 (:method (getBestPrice ?db ?author ?title ?newPrice)
 (forall (?s)
    (shoplist ?s)
    ((in ?s ?db)
    ((call <= ?newPrice ?price)
    (assign ?newPrice ?price)
    )
    )
 )
 (
    (reassignBestPrice ?s ?newPrice ?author ?title)
 )
 )


 (:method (reassignBestPrice ?s ?l ?author ?title)
    ((shoplist (item ?item) (author ?author) (title ?title) (qty ?quantity) (price ?price))
        (lowest (lowestPrice ?lowest))
        (call <= ?price ?lowest)
        (assign ?lowest ?price)
        )
    ( (!displayPrice ?author ?title ?price) (!purchase ?author ?title ?price) )
 )


 (:method (getBestPrice ?db ?author ?title ?newPrice) ; CASE 3 - Strategy 3 (Title misspelt)
 (forall (?s)
    (shoplist ?s)
  ((in ?s ?db)
    ((call <= ?newPrice ?price)
 (assign ?newPrice ?price)
    )
    )
 )
 (
```

```
    ((!incorrectTitle ?title) (reassignBestPrice ?s ?newPrice ?author ?title1))
)
)

(:method (getBestPrice ?db ?author ?title ?newPrice) ; CASE 3 - Strategy 3 (Author misspelt)
(forall (?s)
    (shoplist ?s)
    ((in ?s ?db)
    ((call <= ?newPrice ?price)
    (assign ?newPrice ?price)
    )
   )
)
(
    ((!incorrectAuthor ?author) (reassignBestPrice ?s ?newPrice ?author1 ?title))
)
)

(:method (getBestPrice ?db ?author) ; CASE 3 - Strategy 3 (Missing 2 parameters)
(forall (?s)
    (shoplist ?s)
    ((in ?s ?db)
    ((call <= ?newPrice ?price)
    (assign ?newPrice ?price)
    )
    )
)
(
    ((reassignBestPrice ?s ?newPrice ?author ?title))
)
)

(:method (getBestPriceUnresponsive ?db ?author ?title ?newPrice) ;CASE3 - Startegy 2 (Unresponsive)
(forall (?s)
    (shoplist ?s)
    ((in ?s ?db)
    ((call <= ?newPrice ?price)
    (assign ?newPrice ?price)
    )
    )
)
(
    (getBestPriceUnresponsive ?db ?author ?title ?newPrice)
    (reassignBestPrice ?s ?newPrice ?author ?title)
)
)

(:method (getBestPriceUnresponsive ?db ?author ?title ?newPrice)
(forall (?s)
   (shoplist ?s)
```

```
        ((in ?s ?db)
        ((call <= ?newPrice ?price)
        (assign ?newPrice ?price)
        )
        )
    )
    (
        ((!replan ?author) (reassignBestPrice ?s ?newPrice ?author ?title))
    )
    )

)
)
```

B.2 Problem Definition (JSHOP2)

```
(defproblem problem bookstore
    (

        (shoplist (item Textbook) (author Bishop) (title CSharp) (qty 1) (price 80))
        (shoplist (item Textbook) (author Bishop) (title Java) (qty 1) (price 65))
        (shoplist (item Textbook) (author Bishop) (title DesignPatterns) (qty 3) (price 20))
        (shoplist (item Textbook) (author Bishop) (title Java) (qty 2) (price 75))
        (shoplist (item Fictionbook) (author King) (title TheEyesoftheDragon) (qty 1) (price 50))

        (database db)
        (in (shoplist (item Textbook) (author Bishop) (title DesignPatterns) (qty 3) (price 20)) db)
        (in ((shoplist (item Textbook) (author Bishop) (title Java) (qty 1) (price 65)) db))
        (in((shoplist (item Fictionbook) (author Bishop) (title Java) (qty 1) (price 75)) db))
        (in((shoplist (item Otherbook) (author Bishop) (title CSharp) (qty 1) (price 80)) db))
        (in ((shoplist (item Fictionbook) (author King) (title TheEyesoftheDragon) (qty 1) (price 50)) db))

        (lowest (lowestPrice 50))

        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;;PUT BOOKS INTO A DATABASE (e.g. listat shoplistname, dbname);;
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    )
    (

        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;;;CASE 1: Get the price of the book                              ;;;
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        (getPrice Bishop Java) ;valid input
        (getPrice Bishop) ;Interface change - S3)
        (getPrice Bishop Javaa) ;Workflow Inconsistency
        (getPrice King Java) ;invalid combination of inputs
        (getPriceUnresponsive Bishop Java) ;Time-out error
```

```
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;;;CASE 2: Buy a book                                                  ;;;
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        (buy Bishop Java 70)
        (buy King TheEyesoftheDragon 20)
        (buy Bishop)
        (buy CSharp 80)
        (buy King CShaer 50)
        (buyUnresponsive Bishop Java 100)


        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;;;CASE 3: Get the best price for a book                            ;;;
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;;; The last parameter is uses to specify the largest price allowed
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        (getBestPrice db Bishop Java 100) ;Valid input
        (getBestPrice db King EyesofTheDragon 100) ;valid input
        (getBestPrice db Bishop) ;Interface change - S3)
        (getBestPrice db Bishop Javaa 100) ;Workflow Inconsistency
        (getBestPriceUnresponsive db Bishop Java 100) ;Time-out error

    )
)
```

B.3 Experimental Result Sets (JSHOP2)

B.3.1 Case 1: Get price of a specific item

```
1
2 plan(s) were found:

Plan #1:
Plan cost: 2.0
6
(!displayprice bishop java 65.0)
(!noproblem bishop)
----------------------

11 Plan #2:
Plan cost: 2.0

(!displayprice bishop java 75.0)
(!noproblem bishop)
16 ----------------------
```

Table B.1: Experimental results of Test Case 1 in JSHOP2

Faults Triggered	**Strategies**	**Status**	**Indicator**
Interface changed **(getPrice Bishop Java)**	1	failed	no plan found
	2	failed	no plan found
	3	successful	See Listing B.1
Interface changed **(getPrice Bishop)**	1	failed	no plan found
	2	failed	no plan found
	3	partially failed	Found all the books by Bishop, however, the results return contain two extra books that are not part of the request. (See Listing B.2)
Interface changed **(getPrice Java)**	1	failed	no plan found
	2	failed	no plan found
	3	partially failed	Found all Java books available. (See Listing B.3)
Time-out **(getPriceDelayed Bishop Java)**	1	failed	No plan was found due to unreachable service
	2	failed	No plan was found due to unreachable service. The plan generation was terminated after serveral unsuccessful attempt of re-execution.
	3	successful	Two plans were found. See Listings B.4
Incorrect input (workflow inconsistency) - **(getPrice Bishop Javaa)**	1	failed	no plan found
	2	failed	no plan found
	3	partially failed	Re-planning shows four items. See Listings B.5
Unknown faults -**(getPrice King Java)**	1	failed	no plan found
	2	failed	no plan found
	3	partially failed	Re-planning shows all the results that consisting of the correct author name and the title of the book. See Listings B.6

```
Time Used = 0.016
```

Listing B.1: Plans for Case 1 Interface Changed - `(getPrice Bishop Java)`

```
4 plan(s) were found:

Plan #1:
Plan cost: 2.0

(!displayprice bishop csharp 80.0)
(!noproblem csharp)
----------

Plan #2:
Plan cost: 2.0

(!displayprice bishop java 65.0)
(!noproblem java)
----------

Plan #3:
Plan cost: 2.0

(!displayprice bishop designpatterns 20.0)
(!noproblem designpatterns)
----------

Plan #4:
Plan cost: 2.0

(!displayprice bishop java 75.0)
(!noproblem java)
----------

Time Used = 0.0
```

Listing B.2: Plans for Case 1 Interface Changed - `(getPrice Bishop)`

```

2 plan(s) were found:

Plan #1:
Plan cost: 2.0

(!displayprice bishop java 65.0)
(!noproblem java)
----------
```

```
Plan #2:
Plan cost: 2.0

(!displayprice bishop java 75.0)
(!noproblem java)
————————————————————

Time Used = 0.0
```

Listing B.3: Plans for Case 1 Interface Changed - `(getPrice Bishop)`

```
2 plan(s) were found:

Plan #1:
Plan cost: 3.0

(!noresponse bishop)
(!replan bishop)
(!displayprice bishop java 65.0)
————————————————————

Plan #2:
Plan cost: 3.0

(!noresponse bishop)
(!replan bishop)
(!displayprice bishop java 75.0)
————————————————————

Time Used = 0.0
```

Listing B.4: Plans for Case 1 Time-out - `(getPriceDelayed Bishop Java)`

```
4 plan(s) were found:

Plan #1:
Plan cost: 2.0

(!displayprice bishop csharp 80.0)
(!incorrecttitle javaa)
————————————————————

Plan #2:
Plan cost: 2.0

(!displayprice bishop java 65.0)
(!incorrecttitle javaa)
```

```
_____________________

Plan #3:
Plan cost: 2.0

(!displayprice bishop designpatterns 20.0)
(!incorrecttitle javaa)
_____________________

Plan #4:
Plan cost: 2.0

(!displayprice bishop java 75.0)
(!incorrecttitle javaa)
_____________________

Time Used = 0.0
```

Listing B.5: Plans for Case 1 Time-out - `(getPrice Bishop Javaa)`

```
5 plan(s) were found:

Plan #1:
Plan cost: 2.0

(!displayprice king theeyesofthedragon 50.0)
(!incorrecttitle java)
_____________________

Plan #2:
Plan cost: 2.0

(!displayprice bishop java 65.0)
(!incorrectauthor king)
_____________________

Plan #3:
Plan cost: 2.0

(!displayprice bishop java 65.0)
(!incorrectauthor king)
_____________________

Plan #4:
Plan cost: 2.0

(!displayprice bishop java 75.0)
(!incorrectauthor king)
_____________________
```

```
Plan #5:
Plan cost: 2.0

(!displayprice bishop java 75.0)
(!incorrectauthor king)
----------

Time Used = 0.0
```

Listing B.6: Plans for Case 1 Unknown Fault - `(getPrice King Java)`

B.3.2 Case 2: Purchase an item specified

```
1 plan(s) were found:

Plan #1:
Plan cost: 4.0

(!purchase bishop java 65.0)
(!transferfunds 65.0)
(!balance 5.0)
(!noproblem bishop)
----------

Time Used = 0.0
```

Listing B.7: Plans for Case 2 Interface Changed - `(buy Bishop Java 70)`

```

4 plan(s) were found:

Plan #1:
Plan cost: 3.0

(!purchase bishop csharp 80.0)
(!noproblem bishop)
----------

Plan #2:
Plan cost: 3.0

(!purchase bishop java 65.0)
(!noproblem bishop)
----------

Plan #3:
Plan cost: 3.0

(!purchase bishop designpatterns 20.0)
```

Table B.2: Experimental results of Test Case 2 in JSHOP2

Faults Triggered	**Strategies**	**Status**	**Indicator**
Interface changed **(buy Bishop Java 70)**	1	failed	no plan found
	2	failed	no plan found
	3	successful	See Listing B.7
Interface changed **(buy Bishop)**	1	failed	no plan found
	2	failed	no plan found
	3	successful	Found all the books by Bishop. See Listing B.8
Interface changed **(buy CSharp 80)**	1	failed	no plan found
	2	failed	no plan found
	3	successful	Found all service that have CSharp books available. (See Listing B.9)
Time-out **(buyUnresponsive Bishop Java 70)**	1	failed	No plan was found due to unreachable service
	2	failed	No plan was found due to unreachable service. The plan generation was terminated after serveral unsuccessful attempt of re-execution.
	3	successful	Re-planning binds to other servicew which have the same book available. See Listings B.10
Incorrect input (workflow inconsistency) - **(buy King cshaer 50)**	1	failed	no plan found
	2	failed	no plan found
	3	partially failed	Re-planning shows one book by King. See Listings B.11

```
(!noproblem bishop)
------------------------

Plan #4:
Plan cost: 3.0

(!purchase bishop java 75.0)
(!noproblem bishop)
------------------------

Time Used = 0.0
```

Listing B.8: Plans for Case 2 Interface Changed - `(buy Bishop)`

```
1 plan(s) were found:

Plan #1:
Plan cost: 2.0

(!purchase bishop csharp 80.0)
(!noproblem bishop)
------------------------

Time Used = 0.016
```

Listing B.9: Plans for Case 2 Interface Changed - `(buy CSharp 80)`

```
4 plan(s) were found:

Plan #1:
Plan cost: 6.0

(!noresponse bishop)
(!replan bishop)
(!purchase bishop java 65.0)
(!transferfunds 65.0)
(!balance 35.0)
(!noproblem bishop)
------------------------

Plan #2:
Plan cost: 6.0

(!noresponse bishop)
(!replan bishop)
(!purchase bishop java 75.0)
(!transferfunds 75.0)
```

```
(!balance 25.0)
(!noproblem bishop)
_____________________

Plan #3:
Plan cost: 6.0

(!noresponse bishop)
(!replan bishop)
(!purchase bishop java 65.0)
(!transferfunds 65.0)
(!balance 35.0)
(!noproblem bishop)
_____________________

Plan #4:
Plan cost: 6.0

(!noresponse bishop)
(!replan bishop)
(!purchase bishop java 75.0)
(!transferfunds 75.0)
(!balance 25.0)
(!noproblem bishop)
_____________________

Time Used = 0.015
```

Listing B.10: Plans for Case 2 Time-out - `(buyUnresponsive Bishop Java 70)`

```
1 plan(s) were found:

Plan #1:
Plan cost: 5.0

(!incorrecttitle cshaer)
(!purchase king theeyesofthedragon 50.0)
(!transferfunds 50.0)
(!balance 0.0)
(!noproblem king)
_____________________

Time Used = 0.015
```

Listing B.11: Plans for Case 2 Incorrect Input (Workflow inconsistency) - `(buy King cshaer 50)`

B.3.3 Case 3: Find the best prices and purchases the item

```
3 plan(s) were found:

Plan #1:
Plan cost: 2.0

(!displayprice bishop csharp 45.0)
(!purchase bishop csharp 45.0)
--------------------

Plan #2:
Plan cost: 2.0

(!displayprice bishop java 35.0)
(!purchase bishop java 35.0)
--------------------

Plan #3:
Plan cost: 2.0

(!displayprice bishop designpatterns 20.0)
(!purchase bishop designpatterns 20.0)
--------------------

Time Used = 0.016
```

Listing B.12: Plans for Case 3 Interface Changed - `(getBestPrice db Bishop)`

```
1 plan(s) were found:

Plan #1:
Plan cost: 3.0

(!replan bishop)
(!displayprice bishop java 35.0)
(!purchase bishop java 35.0)
--------------------

Time Used = 0.016
```

Listing B.13: Plans for Case 3 Time-out - `(getBestPriceUnresponsive db Bishop)`

Table B.3: Experimental results of Test Case 3 in JSHOP2

Faults Triggered	**Strategies**	**Status**	**Indicator**
Interface changed (**getBestPrice db Bishop**)	1	failed	no plan found
	2	failed	no plan found
	3	successful	See Listing B.12
Time-out (**getBestPriceUnresponsive Bishop Java 100**)	1	failed	No plan was found due to unreachable service
	2	failed	No plan was found due to unreachable service. The plan generation was terminated after serveral unsuccessful attempt of re-execution.
	3	successful	Re-planning binds to other service which have the same book available. See Listings B.13
Incorrect input (workflow inconsistency) - (**getBestPrice db Bishop Javaa 100**)	1	failed	no plan found
	2	failed	no plan found
	3	partially failed	The system is able to detect the title of the book requested is incorrect. Re-planning shows all the books by the author Bishop. See Listings B.14

```
3 plan(s) were found:

Plan #1:
Plan cost: 3.0

(!incorrecttitle javaa)
(!displayprice bishop csharp 45.0)
(!purchase bishop csharp 45.0)
----------

Plan #2:
Plan cost: 3.0

(!incorrecttitle javaa)
(!displayprice bishop java 35.0)
(!purchase bishop java 35.0)
----------

Plan #3:
Plan cost: 3.0

(!incorrecttitle javaa)
(!displayprice bishop designpatterns 20.0)
(!purchase bishop designpatterns 20.0)
----------

Time Used = 0.015
```

Listing B.14: Plans for Case 3 Incorrect Inputer (Workflow Inconsistency) - `(getBestPrice db Bishop Javaa 100)`

Appendix C

Derived Publications

- K.S. May Chan, Tim Grant and Judith Bishop, ***Self-Healing Web Service Composition Using HTN Planners***, Proceedings of SAICSIT 2006, pp. 268. Somerset West, South Africa (Poster).
- K.S. May Chan, Judith Bishop, Johan Steyn, Luciano Baresi and Sam Guinea, ***A Fault Taxonomy for Web Service Composition***, Proceedings of the Third International Workshop on Engineering Service Oriented Applications (WESOA'07), Springer LNCS. Vienna, Austria, 2007.
- K.S. May Chan and Judith Bishop, ***A Self-healing Composition Cycle for Web Services***, Technical Report, 2008.
- K.S. May Chan, Judith Bishop and Luciano Baresi, ***Survey and Comparison of Planning Techniques for Web Services Composition***, Technical Report, 2008.

Acronyms

AI	Artificial Intelligent
BPEL	Business Process Execution Language
BPML	Business Process Modeling Language
DAML-S	Darpa Agent Markup Language for Services
GOST	Goal Structure Table
HTN	Hierarchical Task Network
JSHOP	Java implementation of Simple Hierarchical Ordered Planner
MAPE	Monitor, Analyzer, Planner, Executive
O-Plan	Open Planning Architecture
OWL	Web Ontology Language
OWL-S	Web Ontology Language for Services
QoS	Quality of Service
SHOP	Simple Hierarchical Ordered Planner
SIPE	System for Interactive Planning and Execution Monitoring
SLA	Service Level Agreement
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SOC	Service-Oriented Computing
TOME	Table of Multiple Effects
UDDI	Universal Description Discovery and Integration
WSDL	Web Service Description Language
WSDL-S	Web Service Description Language with Semantics
XML	Extensible Markup Language
XSD	XML Schema Definition

Bibliography

[1] Marco Aiello, Mikio Aoyama, Francisco Curbera, and Mike P. Papazoglou (eds.). *Hybrid Web Service Composition: Business Processes Meet Business Rules*. ACM, 2004.

[2] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M. Schmidt, A. Sheth, and K. Verma. *Web Service Semantics – WSDL-S*. A joint UGA-IBM Technical Note, version, vol. 1, 2005.

[3] Assaf Arkin. *Business Process Modeling Language*, 2002. URL `http://xml.coverpages.org/BPML-2002.pdf`, last accessed on 18 February 2008.

[4] Muhammad Ahtisham Aslam, Sören Auer, Jun Shen, and Michael Hermann. *Expressing Business Process Models as OWL-S Ontologies*. In *Second International Workshop on Grid and Peer-to-Peer based Workflows*, *Lecture Notes in Computer Science*, vol. 4103. Springer, 2006, pp. 400–415.

[5] Daniel Austin, Abbie Barbir, Christopher Ferris, and Sharad Garg. *Web Services Architecture Requirements*, 2004. W3C Working Group Note, 11 February 2004. W3C (World Wide Web Consortium).

[6] Algirdas Avižienis, Jean-Claude Laprie, Brain Randell, and Carl Landwehr. *Basic Concepts and Taxonomy of Dependable and Secure Computing*. IEEE Transactions on Dependable and Secure Computing, (1), 2004, pp. 11–33.

[7] Luciano Baresi, Carlo Ghezzi, and Sam Guinea. *Smart monitors for composed services*. In Marco Aiello, Mikio Aoyama, Francisco Curbera, and Mike P. Papazoglou (eds.), *ICSOC*. ACM, 2004, pp. 193–202.

[8] Luciano Baresi, Carlo Ghezzi, and Same Guinea. *Towards Self-healing Compositions of Service*. Contributions to Ubiquitous Computing, 2007, pp. 27–46.

[9] Luciano Baresi and Sam Guinea. *Towards Dynamic Monitoring of WS-BPEL Processes*. In Boualem Benatallah, Fabio Casati, and Paolo Traverso (eds.), *ICSOC*, *Lecture Notes in Computer Science*, vol. 3826. Springer, 2005, pp. 269–282.

[10] Tom Bellwood, Steve Capell, Luc Clement, John Colgrave, Matthew J. Dovey, Daniel Feygin, and Andrew Hately. *UDDI*, 2004. URL `http://uddi.org/pubs/uddi\_v3.htm`, last accessed on 18 February 2008.

[11] Boualem Benatallah, Fabio Casati, and Paolo Traverso (eds.). *Template-Based Automated Service Provisioning - Supporting the Agreement-Driven Service Life-Cycle*, Lecture Notes in Computer Science. Springer, 2005.

[12] Tim Berners-Lee, James Hendler, and Ora Lassila. *The Semantic Web*. Scientific American, vol. 284(5), 2001, pp. 34–43.

[13] D. Chakraborty and A. Joshi. *Dynamic Service Composition: State-of-the-Art and Research Directions*. Baltimore County, Baltimore, USA, 2001. Technical Report TR-CS-01-19.

[14] D. Chakraborty, F. Perich, A. Joshi, T. Finin, and Y. Yesha. *A Reactive Service Composition Architecture for Pervasive Computing Environments*. In *7th Personal Wireless Communications Conference (PWC 2002)*. 2002.

[15] K.S. May Chan, Judith Bishop, and Luciano Baresi. *Survey and Comparison of AI Planning Techniques in Web Service Composition*. Technical Report, 23 February 2007.

[16] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) 1.1*, 2001. URL `http://www.w3.org/TR/wsdl`, last accessed on 18 February 2008.

[17] Ken Currie and Austin Tate. *O-Plan: the Open Planning Architecture*. Artificial Intelligence, vol. 52(1), 1991, pp. 49–86.

[18] Prashant Doshi, Richard Goodwin, Rama Akkiraju, and Kunal Verma. *Dynamic Workflow Composition: Using Markov Decision Processes*. International Journal of Web Service Research, vol. 2(1), 2005, pp. 1–17.

[19] Brian Drabble, Austin Tate, and Jeff Dalton. *Repairing Plans On-the-fly*. In *NASA Workshop on Planning and Scheduling for Space*. Oxnard, California, USA, 1997.

[20] Dieter Fensel, Katia P. Sycara, and John Mylopoulos (eds.). *Adapting BPEL4WS for the Semanitc Web: The Bottom-Up Approach to Web Service Interoperation*, *Lecture Notes in Computer Science*, vol. 2870. Springer, 2003.

[21] Thomas Friese, Jörg P. Müller, and Bernd Freisleben. *Self-healing Execution of Business Processes Based on a Peer-to-Peer Service Architecture*. In *ARCS*. 2005, pp. 108–123.

[22] David Garlan and Bradley Schmerl. *Model-based adaptation for self-healing systems*. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*. ACM Press, New York, NY, USA, 2002, pp. 27–32.

[23] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufman Publishers, San Francisco, California, USA, 2004.

[24] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. *SOAP Version 1.2*, 2003. URL `http://www.w3.org/TR/soap12-part1/`, last accessed on 18 February 2008.

[25] Sherif A. Gurguis and Amir Zeid. *Towards autonomic web services: achieving self-healing using web services*. ACM SIGSOFT Software Engineering Notes Archive, vol. 30(4), 2005, pp. 1–5.

[26] IBM. *Automating problem determination: A first step toward self-healing computing systems*, 2003. IBM White Paper.

[27] Okhtay Ilghami. *Documentation for JSHOP2*, 2006. Technical Report CS-TR-4694.

[28] Leslie Lamport. *Concurrent Reading and Writing of Clocks.* ACM Trans. Comput. Syst., vol. 8(4), 1990, pp. 305–310.

[29] Therani Madhusudan and N. Uttamsingh. *A declarative approach to composing web services in dynamic environments*. Decis. Support Syst., vol. 41(2), 2006, pp. 325–357.

[30] David Martin, Anupriya Ankolekar, Mark Burstein, Grit Denker, Daniel Elenius, Jerry Hobbs, Lalana Kagal, Ora Lassila, Drew McDermott, Deborah McGuinness, Sheila McIlraith, Massimo Paolucci, Bijan Parsia, Terry Payne, Marta Sabou, Craig Schlenoff, Evren Sirin, Monika Solanki, Naveen Srinivasan, Katia Sycara, and Randy Washington. *OWL-S 1.1 Release*, 2004. URL `http://www.daml.org/services/owl-s/1.1/`, last accessed on 18 February 2008.

[31] Drew V. McDermott. *Estimated-Regression Planning for Interactions with Web Services*. In *Sixth International Conference on Artificial Intelligence Planning Systems*. AAAI, 2002, pp. 204–211.

[32] Deborah L. McGuinness and Frank van Harmelen. *OWL Web Ontology Language Overview*, 2004. URL `http://www.w3.org/TR/owl-features/`, last accessed on 19 February 2008.

[33] Sheila McIlraith and Tran Cao Son. *Adapting Golog for Composition of Semantic Web Services*. In *8th International Conference on Principles and Knowledge Representation and Reasoning*. Morgan Kaufmann, 2002, pp. 482–496.

[34] Brahim Medjahed, Athman Bouguettaya, and Ahmed K. Elmagarmid. *Composing Web services on the Semantic Web.* VLDB J., vol. 12(4), 2003, pp. 333–351.

[35] Marija Mikic-Rakic, Nikunj Mehta, and Nenad Medvidovic. *Architectural style requirements for self-healing systems.* In *WOSS '02: Proceedings of the first workshop on Self-healing systems.* ACM Press, New York, NY, USA, 2002, pp. 49–54.

[36] Henri Naccache and Gerald C. Gannod. *A Self-Healing Framework for Web Services.* In *2007 IEEE International Conference on Web Services (ICWS).* IEEE Computer Society, 2007, pp. 398–345.

[37] Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. *SHOP2: An HTN Planning System.* Journal of Artificial Intelligence Research, 2003, pp. 379–404.

[38] OASIS. *Web Services Business Process Execution Language Version 2.0*, 2007. URL `http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.pdf`, last accessed on 02 October 2007.

[39] Oracle. *Oracle BPEL Process Manager Suite 10g.* URL `http://www.oracle.com/technology/products/ias/bpel/index.html`, last accessed on 19 February 2008.

[40] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. *An Architecture-Based Approach to Self-Adaptive Software.* IEEE Intelligent Systems, vol. 14(3), 1999, pp. 54–62.

[41] Massimo Paolucci, Naveen Srinivasan, Katia P. Sycara, and Takuya Nishimura. *Towards a Semantic Choreography of Web Services: From WSDL to DAML-S.* In *Proceedings of the International Conference on Web Services.* CSREA Press, 2003, pp. 22–26.

[42] M.P. Papazoglou and D. Georgakopoulos. *Service-Oriented Computing.* Communication of the ACM, vol. 46(10), 2003, pp. 25–28.

[43] Joachim Peer. *Web Service Composition as AI Planning - a Survey*, 2005. URL `http://elektra.mcm.unisg.ch/pbwsc/docs/pfwsc.pdf`, last accessed on 19 February 2008.

[44] Randall Perrey and Mark Lycett. *Service-Oriented Architecture*. In *2003 Symposium on Applications and the Internet Workshops (SAINT 2003)*. IEEE Computer Society, 2003, pp. 116–119.

[45] Marco Pistore, Fabio Barbon, Piergiorgio Bertoli, D. Shaparau, and Paolo Traverso. *Planning and Monitoring Web Service Composition*. 2004.

[46] Marco Pistore, Paolo Traverso, and Piergiorgio Bertoli. *Automated Composition of Web Services by Planning in Asynchronous Domains*. In *15th International Conference on Automated Planning and Scheduling*. The AAAI Press, 2005, pp. 2–11.

[47] Jun Shen, Yun Yang, Chengang Wan, and Chuan Zhu. *From BPEL4WS to OWL-S: Integrating E-Business Process Descriptions*. In *SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing*. IEEE Computer Society, Washington, DC, USA, 2005, pp. 181–190.

[48] E. Sirin and B. Parsia. *Planning for Semantic Web Services*, 2004. URL `http://citeseer.ist.psu.edu/sirin04planning.html`, last accessed on 19 February 2008.

[49] Evren Sirin, Bijan Parsia, Dan Wu, James Hendler, and Dana Nau. *HTN Planning for Web Service Composition Using SHOP2*. Journal of Web Semantics, vol. 1, 2004, pp. 377–396.

[50] Austin Tate and Brian Drabble. *O-Plan - Architecture Guide Version 2.3*, 1995. URL `citeseer.ist.psu.edu/tate95oplan.html`, last accessed on 17 February 2008.

[51] David E. Wilkins. *SIPE-2 Architecture*. URL `http://www.ai.sri.com/~sipe/architecture.html`, last accessed on 19 February 2008.

[52] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufman Publishers, San Mateo, California, USA, 1988.

[53] David E. Wilkins. *SIPE-2: System for Interactive Planning and Execution*, 2000. URL `http://www.ai.sri.com/~sipe/`, last accessed on 19 February 2008.

[54] David E. Wilkins and Marie desJardins. *A call for knowledge-based planning*, 2000. URL `citeseer.ist.psu.edu/article/wilkins00call.html`, last accessed on 19 February 2008.

[55] WSDL2OWLS. URL `http://www.daml.ri.cmu.edu/wsdl2owls`, last accessed on 19 February 2008.

[56] Tao Yu and Kwei-Jay Lin. *Service Selection Algorithms for Composing Complex Services with Multiple QoS Constraints*. In *Thired International Conference on Service-Oriented Computing (ICSOC 2005)*, *Lecture Notes in Computer Science*, vol. 3826. Springer, 2005, pp. 130–143.