Automated Web Service Composition

Dónal Murtagh

A dissertation submitted to the University of Dublin,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

2004

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: ___________________
Dónal Murtagh
13 September 2004

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: ____________________
Dónal Murtagh
13 September 2004

Abstract

Web content has traditionally been dominated by documents intended for human interpretation. However, the recent advent of Web Services has played a key role in evolving the Web from a mere document repository to a provider of services. The range of services available may be further expanded by composing Web Services together to create new services which provide novel functionality. The ability of Web Service composition to create new services which are capable of fulfilling a need – that no single service can satisfy – is its major benefit. Web Service composition also has the potential to make Web Service invocation more efficient by aggregating disparate processes into a single service.

The full potential of Web Service composition can only be realised if the process is automated and performed on-demand. Specific advantages of automated service composition over static service composition include its ability to:

- Take advantage of new services as they become available
- Take account of changing user preferences
- Cope with service unavailability

In order to automate service composition, services must describe their characteristics in an unambiguous machine-readable format. In other words, it must be possible for a machine to determine what those services *actually do*. The current standard for encoding such descriptions is OWL-S. In order to solve a Web Service composition problem a machine must also be able to determine which services to execute, and in what order. This problem can be solved by AI planning.

The Web Services composition solution presented in this report reasons over Web Service described in OWL-S and uses a type of planning known as HTN planning in order to specify the composition.

Acknowledgements

Thanks of the first-order are due to Declan O'Sullivan and Dave Lewis, who supervised this dissertation with indefatigable enthusiasm. The successes of this work owe much to their guidance and encouragement; the failings are entirely my own.

Sincere gratitude is also due to those who developed and made freely available various tools that were used during the course of this work. Many others contributed by responding to my e-mails and posts to discussion lists. I am particularly indebted on this front to Bijan Parsia, Evren Sirin, Fusun Yaman, Mithun Sheshagiri, and Okhtay Ilghami.

Table of Contents

Table of Figures viii
1 INTRODUCTION 1
1.1 Web Service Composition 1
1.2 Objectives 3
1.3 Roadmap 5
2 BACKGROUND 6
2.1 Overview 6
2.2 WSDL 6
2.2.1 WSDL Introduction 6
2.2.2 WSDL Example 7
2.3 OWL-S 8
2.3.1 OWL-S Introduction 8
2.3.2 ServiceProfile 10
2.3.3 ServiceModel 11
2.3.4 ServiceGrounding 13
2.4 AI Planning 13
2.4.1 Introduction 13
2.4.2 Classical Planning 14
2.4.3 HTN Planning 17
2.4.4 Suitability of HTN Planning to Web Service Composition 19
3 DESIGN & IMPLEMENTATION 21
3.1 Requirements 21
3.2 Design & Implementation Technologies 22
3.2.1 JESS 22
3.2.1.1 JESS Facts 22
3.2.1.2 JESS Rules 23
3.2.2 Jena 23
3.2.3 OWLJessKB 23
3.2.4 OWL-S API 24
3.2.5 Pellet 24
3.2.6 JSHOP 24
3.2.6.1 Introduction 24
3.2.6.2 JSHOP Formalism 25
3.2.6.2.1 JSHOP Domain Definition 25
3.2.6.2.2 JSHOP Problem Definition 27
3.2.6.3 JSHOP Example 27
3.3 OWLS2JSHOP Architecture 28
3.4 Design & Implementation Issues 30
3.4.1 Choosing the Planner 30
3.4.2 State Representation 31
3.4.3 Conditional Expressions 32
3.5 Component Interactions 34
3.6 OWLS2JSHOP Package 36
3.6.1 Planner 38
3.6.2 OWL2JESS 38
3.6.3 DataFlowManager 38
3.6.4 OntologyOracle 39

3.6.5 SHOPElement 40
3.6.6 SHOPAtomicProcess 40
3.6.7 SHOPAtomicProcessWithPostconditions 40
3.6.8 SHOPCompositeProcess 40
3.6.9 SHOPCompositeProcessSequence 40
4 SCENARIO 41
4.1 Introduction 41
4.2 Scenario Inputs 41
4.3 Scenario Domain Definition 45
4.4 Scenario Problem Description 49
4.5 Scenario Solution 50
5 RELATED WORK 51
5.1 Partial-Order Planning 51
5.2 Planning and Execution 52
5.3 Planning with Incomplete Information 54
5.4 State Representation 54
5.5 Complex Action Planning 55
5.6 Composing Web Services without Semantic Descriptions 56
6 CONCLUSIONS 58
6.1 Evaluation 58
6.1.1 AI Planning Type 58
6.1.2 State Representation 58
6.1.3 Conditional Expressions 59
6.1.4 OWLS2JSHOP Evaluation 59
6.1.5 Other Contributions 62
6.2 Future Challenges 62
6.2.1 Service Discovery 62
6.2.2 Scalability 63
6.2.3 OWL-S Issues 63
7 BIBLIOGRAPHY 65
8 APPENDICES 69
8.1 Appendix A 69
8.1.1 BookPriceProcess 69
8.1.2 BookFinderProcess 72
8.1.3 BNPriceProcess 73
8.1.4 CurrencyConverterProcess 74
8.2 Appendix B 77

Table of Figures

Figure 2.1: Major Elements of a WSDL Document

Figure 2.2: Example WSDL Document

Figure 2.3: Top Level of the Service Ontology

Figure 2.4: Top Level of the Process Ontology

Figure 2.5: Composite Process Decomposed to a Sequence of Atomic Processes

Figure 2.6: An Action Schema

Figure 2.7: A Planning Problem

Figure 2.8: Planning Problem Solution

Figure 2.9: Six Corresponding Linearizations into Total-Order Plans

Figure 2.10: HTN House-Building Problem

Figure 2.11: Solution to HTN House-Building Problem

Figure 3.1: Examples of JSHOP operators

Figure 3.2: Example of a JSHOP Method

Figure 3.3: Complete JSHOP Example

Figure 3.4: JSHOP Solution

Figure 3.5: OWLS2JSHOP Architecture

Figure 3.6: Sequence Diagram

Figure 3.7: Core classes in OWLS2JSHOP package

Figure 3.8: OWL-S Parameter Mappings

Figure 4.1: Type::Name of ServiceModel Inputs and Outputs

Figure 4.2: Type Hierarchy of Input and Output Classes

Figure 4.3: Scenario Initial State

Figure 4.4: Conditional Expressions Encoded as Conjunctions of JESS Atoms in OWL-S

Figure 4.5: Scenario Domain Definition

Figure 4.6: Scenario Problem Definition

Figure 4.7: Scenario Solution

1 INTRODUCTION

1.1 Web Service Composition

As Web Services proliferate it becomes increasingly difficult to find a service that can perform the required task. Frequently, there exists no single service capable of performing the task, but there are combinations of services that can. Combining services to meet a need in this manner is what is meant by service composition. Automating this process is the subject of this dissertation.

A motivating example of service composition is that of converting a document from one format to another, for example, from .doc to .pdf. Assume there is no service capable of performing this conversion, but there exists a service which can convert from .doc to an intermediary format (e.g. .rtf) and another that can convert from the intermediary format to .pdf. By invoking these services in the correct sequence the goal can be achieved, despite the absence of a service that explicitly performs the required conversion. The ability to dynamically create new services from existing services is a key benefit of service composition.

The improvement in efficiency brought about by interacting with an aggregation of services is a further benefit of service composition. Consider the example of booking a holiday, and assume there are disparate services which provide car hire, hotel booking, flight reservation, etc. Without service composition, the user (or agent) must provide their name, address, credit card details etc. to each service separately, but if these disparate services could be combined to form a single holiday booking service, this duplicated effort would be eliminated. Furthermore, the process of booking a holiday becomes a single operation, thus eliminating the possibility of booking a hotel and then being unable to obtain a flight (or vice versa), which exists when each service must be executed separately.

Current approaches to Web Service interoperation such as BPEL4WS [BPEL4WS] require that Web Services be composed manually. However, manual composition of

Web Services is inefficient and at the scale of the Internet, impractical. Returning to the document conversion example, if the composition is performed manually, the user must specify a priori what intermediary format the document must take, attempt to locate a service which converts from .doc to this format, and from this format to .pdf. If appropriate services cannot be found, the user must choose a different intermediary format, and make another attempt at locating appropriate services. The inefficiencies of this approach are obvious. For a start, the user should not have to specify the intermediary format in advance – any will suffice – and this strategy will fail if the only means of converting from .doc to .pdf requires the document to assume more than one intermediary format. Ideally, the user should only have to specify their objective – converting from .doc to .pdf – to a Web Service composition agent, which will locate the necessary services and determine the order in which they should be invoked. In addition to locating the relevant services and determining the order in which they should be executed, automated service composition has other advantages over manual service composition:

- Automated service composition can take advantage of new services. If the composition is performed dynamically, the services that are involved may change over time to include new services which achieve the goal faster, cheaper, or to a higher standard.
- Automated service composition can take account of changing criteria (e.g. preferred method of payment) by adjusting the composition to find services which better fit the new requirements.
- Automated service composition can cope with service unavailability. If a service in the composition becomes unavailable an alternative composition may be found which doesn't require the unavailable service

Fully automated Web Service composition can be characterised as a process involving the following three steps:

1. Automated Discovery – locating candidate Web Services for composition;
2. Automated Composition – determining which Web Services to execute, and in what sequence, in order to achieve the stated goal;

3. Automated Execution – invoking the Web Services.

The remainder of this dissertation will focus almost exclusively on the second of these three steps.

In order to automate the process of service composition, it must be possible for a machine to determine what a service actually *does*. This can be achieved by annotating the service with machine-interpretable semantics which describe the service's characteristics. The current standard for such annotations is OWL-S 1.0, which is defined:

"An OWL-based Web service ontology, which supplies Web service providers with a core set of markup language constructs for describing the properties and capabilities of their Web services in unambiguous, computer-interpretable form" [OWL-S 1.0]

In addition to being able to determine what a Web Service does, in order to solve a Web Service composition problem a machine must be able to determine which services to execute, and in what order. In other words, it must decide which sequence of services to execute in order to achieve the goal state (.pdf, in the document conversion problem) from the initial state (.doc). This is *precisely* the type of problem which Artificial Intelligence (AI) planning addresses. In fact, a Web Service composition problem is essentially an AI planning problem. A more detailed description of AI planning and OWL-S is provided in the next chapter.

1.2 Objectives

The main objective of this research was to formulate and implement an approach to composing Web Services described in OWL-S. Once AI planning emerged as a promising candidate approach, the following more specific objectives were identified:

1. Determine the most appropriate type of AI planning with regard to Web Service composition.

There exist several different types of AI planning (e.g. HTN, POP). Choosing the most appropriate is crucial to the success of this project

2. Determine an effective means of representing the world/problem state

   In classical AI planning, state is represented as a set of ground literals in first-order language, but the state of an OWL-S Web Service composition problem is an OWL Knowledge Base (KB). An effective solution must represent state in a manner that preserves the semantics of OWL, but is compatible with AI planning.

3. Identify a means of expressing conditional statements

   In the context of AI planning, an action's preconditions are statements about the state which must be true in order for the action to be applicable. In the context of OWL-S, process preconditions have a similar interpretation. However, OWL-S 1.0 and its predecessors have failed to specify a particular language for encoding preconditions. In order to apply AI planning to the problem of Web Service composition, a means of expressing preconditions in OWL-S must be identified.

4. Identify a means of evaluating conditional statements

   Merely identifying a means of *expressing* conditional statements would be fairly pointless. In order to implement an AI planning solution to Web Service composition, it is necessary that these conditional statements can be *evaluated* by querying the world state.

5. Demonstrate the solution

   Develop a Web Service composition tool which implements the solutions to objectives (1) – (4).

1.3 Roadmap

A synopsis of the material covered in the remaining chapters follows:

- Chapter 2 (Background) provides an introduction to WSDL, OWL-S, and AI planning. A familiarity with these subjects is required in order to understand the Web Services composition solution presented in this report. The latter will be referred to as OWLS2JSHOP hereafter
- Chapter 3 (Design & Implementation) describes the architecture and technologies of OWLS2JSHOP. Issues that arose during the design and implementation of this system are discussed in this chapter.
- Chapter 4 (Scenario) presents a detailed use case of OWLS2JSHOP
- Chapter 5 (Related Work) describes other attempts at composing Web Services and compares them to OWLS2JSHOP
- Chapter 6 (Conclusions) criticises the contributions of this research, and discusses various challenges that face Web Service composition

2 BACKGROUND

2.1 Overview

This chapter provides background information on various topics related to Web Service composition. Specifically, this chapter consists of an introduction to the Web Services Description Language (WSDL), OWL-S, AI Planning, and discusses their relevancy to OWLS2JSHOP.

2.2 WSDL

2.2.1 WSDL Introduction

This section provides an introduction to the Web Services Description Language (WSDL). The WSDL example and some of the explanations have been adapted from [WSDL].

Fundamentally, a Web Service is characterised as follows:

- SOAP (XML over HTTP), is used to communicate between the Web Service and the Web Client;
- The Web Service interface is described in a WSDL document.

A WSDL file is an XML document that completely describes a Web Service. Specifically, a WSDL document specifies:

- *What* the Web Service consists of – types, messages, operations;
- *How* the Web Service is bound to a set of concrete protocols;
- *Where* the Web Service is implemented – host, port.

A WSDL document defines a web service using the elements described in Figure 2.1 below. A WSDL document can also contain other elements, like extension elements and a `service` element that makes it possible to group together the definitions of several web services in a single WSDL document. A complete WSDL syntax overview is available at [WSDL 1.2].

Element	Description
`<portType>`	Defines the operations performed by the Web Service, and the messages involved.
`<message>`	A message is a data element of an operation. Each message consists of one or more parts, and each part has a data type
`<types>`	Defines the data types of the message parts
`<binding>`	Defines the communication protocols used by the Web Service

Figure 2.1: Major Elements of a WSDL Document

2.2.2 WSDL Example

Figure 2.2 shows a fragment of a simplified WSDL document (namespace details have been omitted for the sake of readability).

```
<message name="getTermRequest">
   <part name="term" type="xs:string"/>
</message>
<message name="getTermResponse">
   <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
      <input message="getTermRequest"/>
      <output message="getTermResponse"/>
  </operation>
</portType>

<binding type="glossaryTerms" name="b1">
<soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation>
    <soap:operation
     soapAction="http://example.com/getTerm"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
```

Figure 2.2: Example WSDL Document

This document describes a Web Service `glossaryTerms` that exposes a single request-response `operation` `getTerm`. This operation has an `input message` named `getTermRequest` (a request message sent from client to server), and an `output message` named `getTermResponse` (a response message sent from server to client). The `message` elements define the parts of each message, and the associated data types.

The `binding` element defines the message format and protocol details. This element has two attributes, `name` and `type`. The former defines the name of the binding, and the latter points to the port for the binding, which in this case is `glossaryTerms`. The `soap:binding` element has two attributes, `style` and `transport`. The style attribute can be either `rpc` or `document` – in this case the latter is chosen. The `transport` attribute defines the SOAP protocol to use – in this case HTTP is specified. The `operation` element defines each operation that the port exposes. For each operation the corresponding `soapAction` action must be defined. The input and output encoding must also be specified – `literal` is used here.

2.3 OWL-S

2.3.1 OWL-S Introduction

Although a WSDL document describes a Web Service's interface, protocol and endpoint, it does not provide any *machine-interpretable* information about what the Web Service actually *does*. For example, assume a Web Service exposes two operations, one of which takes a number x, and returns e^x, and the other takes a credit card number x, and returns e^x, the amount in Euro debited from your account. By referring to the WSDL document, a human reader might correctly *guess* which operation is which from the operation names or message names, but a machine would be unable to distinguish between them, as they both have the same number of input and output messages, message parts, and message part types.

Clearly, for the purpose of automated Web Service composition, it must be possible for a machine to distinguish between these two. More generally, in order to enable

Web Service composition, services must describe in a format that is unambiguous and machine-readable, their:

- Properties and Capabilities – for automated discovery
- Preconditions, Effects, Inputs, and Outputs – for automated composition
- Interfaces, Endpoints, and Protocols – for automated execution

The current standard for annotating Web Services with such a description is OWL-S 1.0[1], defined as:

"An OWL-based Web service ontology, which supplies Web service providers with a core set of markup language constructs for describing the properties and capabilities of their Web services in unambiguous, computer-interpretable form" [OWL-S 1.0]

The term *ontology* as it is understood by computer scientists is: *"a formal specification of a shared conceptualization"* [Borst]. *Conceptualisation* refers to an abstract model of phenomena in the world which identifies their relevant characteristics. *Formal* means that the ontology has a well defined semantics, and s*hared* reflects the notion that an ontology captures consensual knowledge, i.e. knowledge which is generally accepted.

OWL, the Web Ontology Language [OWL], is an XML-based language used to describe ontologies. In practical terms, an OWL ontology consists of a set of classes, instances and properties which describe the concepts within a particular domain. Although OWL classes, instances and properties are analogous to those of object orientation (OO), OWL provides a much richer range of constructs for specifying the relationships between classes and properties than OO, which typically only facilitates parent-child relationships. An example OWL ontology describing currencies is available at [Currency].

[1] The current full release is OWL-S 1.0 and all references to OWL-S refer to this version unless otherwise stated. Previous releases of this language were known as DAML-S, and were built upon DAML+OIL (the predecessor of OWL).

The term *OWL-based Web service ontology* refers to the fact that OWL-S is an ontology which provides OWL classes and properties suitable for describing the semantics of Web Services. An OWL-S description for a *particular* Web Service consists of an OWL-S document that instantiates these classes with appropriate property values. OWL-S is expressed in XML syntax, but due to its verbosity, it will be represented diagrammatically wherever possible in this report. Figure 2.3 shows the parent classes of which OWL-S is composed. The nodes correspond to OWL classes and the arcs to OWL properties.

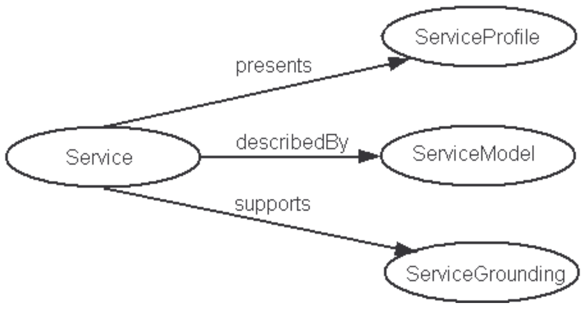


Figure 2.3: Top Level of the Service Ontology

Although the `Service` class provides little by way of description, it acts as an organisational point of reference for OWL-S descriptions. Each Web Service description will provide a single instance of Service, with appropriate property values for `presents`, `describedBy`, and `supports`. The respective ranges of these properties are `ServiceProfile`, `ServiceModel`, and `ServiceGrounding`, each of which provides a different type of knowledge about the service being described. An introduction to each is provided in the following subsections.

2.3.2 ServiceProfile

The service presents a `ServiceProfile`, which has a subclass `Profile`. The `Profile` provides a vocabulary to characterise both functional (e.g. inputs, outputs, preconditions, effecs) and non-functional (e.g. contact information, service category) properties of the service. The `Profile` advertises the services capabilities, and facilitates automated discovery, but is largely irrelevant to composition.

2.3.3 ServiceModel

The service is `describedBy` a `ServiceModel`, which has a subclass `Process`. Figure 2.4 below shows the `Process` class, its subclasses, and its relationship with the `Profile`.

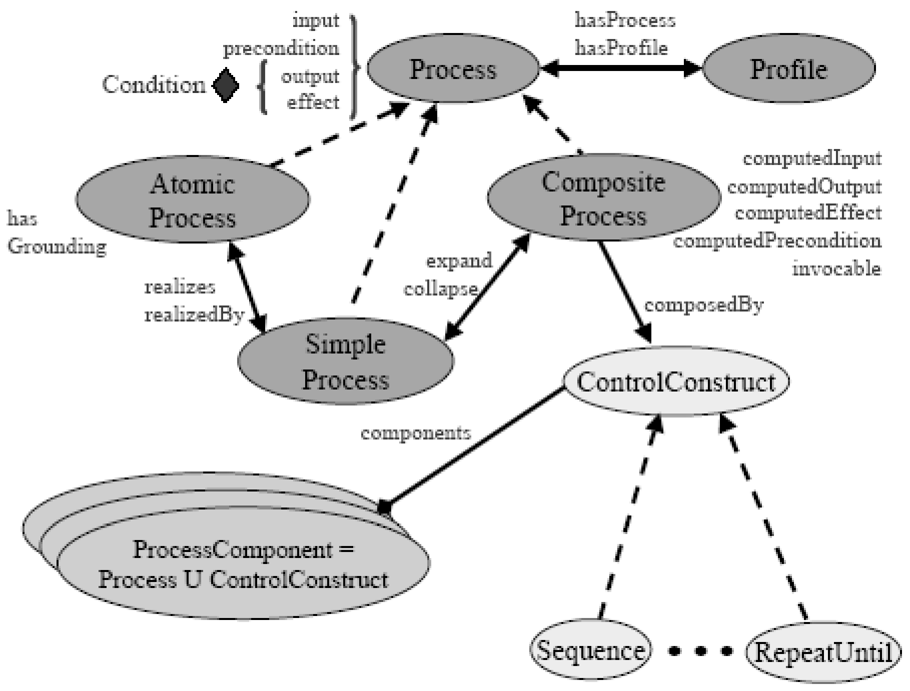


Figure 2.4: Top Level of the Process Ontology

The `Process` class and its subclasses specify how a service works. Specifically, they describe which atomic/composite processes are invoked when a service is executed, their order of invocation, and their inputs, outputs, preconditions, and effects (IOPEs). As such, for the purpose of Web Service composition, the `ServiceModel` is the key part of OWL-S.

An atomic process is one which has no subprocesses and is directly invocable, whereas a composite process can be decomposed into other (atomic or composite) processes[2]. The order in which a composite processes' subprocesses should be executed is specified via a `ControlConstruct` such as: `Sequence`, `Unordered`, `If-`

[2] In fact, there exists a third type of process, a `SimpleProcess`. However, for the purpose of Web Service composition they are *"neither necessary, nor convenient, nor useful"*[HTN '04], and therefore, will be ignored hereafter.

`Then-Else`, `Repeat-Until` [OWL-S 1.0]. For example, Figure 2.5 shows a composite process `BookPriceProcess` which decomposes to a sequence of the atomic processes: `BookFinderProcess`, `BNPriceProcess`, `CurrencyConverterProcess`. In other words, when BookPriceProcess is executed, it in turn executes these atomic processes, in the order stated.

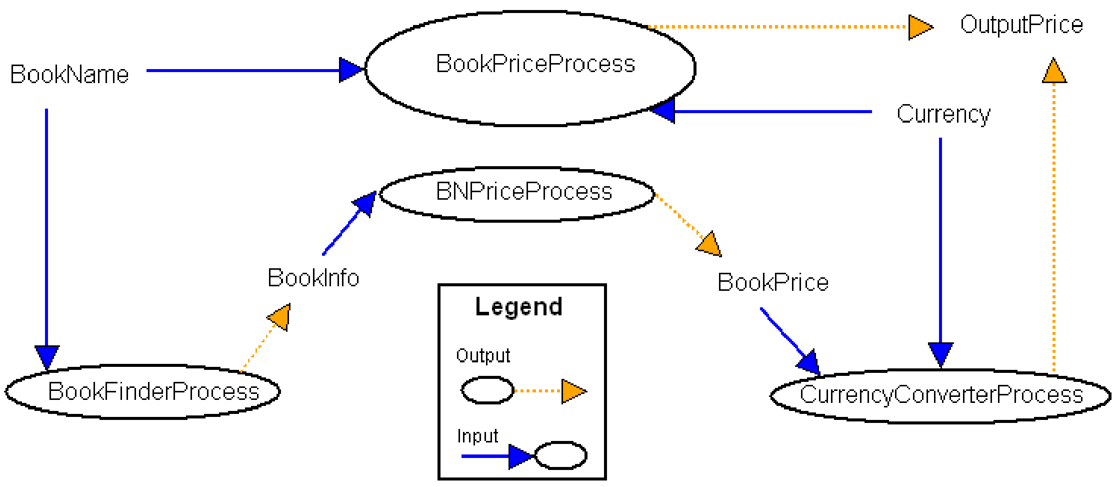


Figure 2.5: Composite Process Decomposed to a Sequence of Atomic Processes

Figure 2.5 also shows the inputs and outputs of each process, and how they relate to one another. Inputs and outputs can be classified as either *internal* or *external* [PLAN '03]. An internal input is one which is provided by the output of another process (e.g. `BookInfo`), whereas an external input must be provided by the user or agent invoking the service (e.g. `BookName`). Similarly, an internal output is one which is supplied to another process (e.g. `BookInfo`), whereas an external output is one which is returned to the user/agent that invoked the service (e.g. `OutputPrice`). This mapping between process parameters is specified in the `ServiceModel` by the data flow model.

In addition to inputs and outputs, processes also have preconditions and effects. While inputs and outputs specify the information that is required and produced by process execution, preconditions specify conditions that must be satisfied for a process to execute and effects represent changes (to the world) resulting from execution. In the context of a bookselling service, a typical precondition is that the items must be in stock, and an effect is that the items purchased are shipped. Examples of an input and output of such a service are the delivery address and a receipt respectively.

Effects and outputs may be either conditional or unconditional. Unconditional outputs/effects are those which are produced for every execution of the process, whereas conditional effects and outputs are only produced under certain conditions. The `ServiceModel` defines the classes `ConditionalOutput` and `ConditionalEffect` which allow a number of conditions to be associated with an output and effect respectively. Although conditions pervade OWL-S – in addition to preconditions and conditional effects/outputs, they also feature in control constructs such as `If-Then-Else` – *"OWL-S 1.0 does not mandate any language for expressing conditions at present, leaving to the modeller the task of deciding which rule language to adopt. Nor does OWL-S 1.0 make expressivity claims or mandate a specific kind of logic for the rule language"* [OWL-S 1.0].

2.3.4 ServiceGrounding

The service supports a `ServiceGrounding`, which has a subclass `Grounding`. This class provides a mapping from the abstract OWL-S `ServiceModel` to a concrete specification. The concrete specification is invariably a WSDL description, in which case the `ServiceGrounding` class maps:

- OWL Classes to WSDL types
- Process inputs/outputs to WSDL input/output message parts
- Atomic processes to WSDL operations

The `ServiceGrounding` facilitates automated invocation, but is largely irrelevant to automated composition.

2.4 AI Planning

2.4.1 Introduction

This section discusses the relevancy of AI planning to Web Service composition, and provides an introduction to AI planning in general, and Hierarchical Task Network

(HTN) planning in particular. Much of the material in this chapter is taken from [AI Mod].

2.4.2 Classical Planning

A classical AI planning problem has the following elements:

- An initial state
- A goal state
- A set of actions[3]

The objective of AI planning is to find a plan, that is, a sequence of actions which will achieve the goal state from the initial state. State is represented as a set of ground literals in first-order language, and an action is an expression which has a:

- Name
- Parameter List
- Precondition List – the first-order literals which must belong to the state in order for the action to be applicable
- Effects List – this is composed of an *add list* and a *delete list*. The former is a list of positive effects, which are effects that the action adds to the state. The delete list is a list of negative effects, which are effects that the action removes from the state

A description of an action named `Fly` which flies `p` from `orig` to `dest` is shown in Figure 2.6:

```
ACTION:      (Fly(p, orig, dest)
PRECOND:     At(p, orig) ∧ Plane(p) ∧ Airport(orig) ∧ Airport(dest)
EFFECT:      ¬At(p, orig) ∧ At(p, dest))
```

Figure 2.6: An Action Schema

The parameter list of this action is `(p, orig, dest)`. The precondition list can be interpreted as: `p` must be `At orig`, and `p` must be a `Plane`, and `orig` and `dest` must

[3] Also known as 'operators', both terms will be used in this report.

both be an `Airport`. The '¬' indicates that the first effect in the list is a negative effect, thus the effects of this action can be interpreted as: `p` is not at `orig`, and `p` is at `dest`.

A simple planning problem is shown in Figure 2.7 below. Recall that a planning problem is simply a set of actions, and an initial and goal state. In this trivial example, the initial state is empty and the actions have no parameters.

```
GOAL:        (RightShoeOn ∧ LeftShoeOn)
INIT:        ()
ACTION:      (RightShoe() PRECOND: RightSockOn   EFFECT: RightShoeOn)
ACTION:      (RightSock() EFFECT:  RightSockOn)
ACTION:      (LeftShoe()  PRECOND: LeftSockOn;   EFFECT: LeftShoeOn)
ACTION:      (LeftSock()  EFFECT:  LeftSockOn)
```

Figure 2.7: A Planning Problem

The solution to this problem is illustrated in Figure 2.8 below.

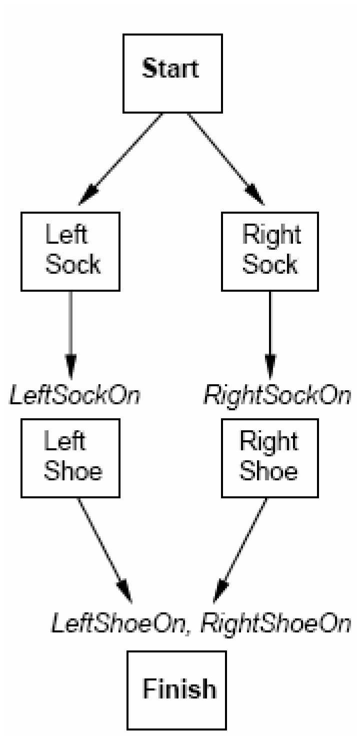


Figure 2.8: Planning Problem Solution

The state after each action is added to the plan is indicated by the literals in italics. For example, the initial state is empty, but after the `RightSock` action, `RightSockOn` is added to the state (alternatively, "RightSockOn is true"). This satisfies the precondition of the action `RightShoe`, thereby enabling it to be added to the plan. The final state is (`LeftShoeOn ∧ RightShoeOn`), which satisfies the goal. In other words, the plan shown in Figure 2.8 is a valid solution to the problem given in Figure 2.7.

The solution shown in Figure 2.8 is in fact a *partial-order* plan, because the sequence in which the actions must occur is not fully specified. For example, although `RightSock` must occur before `RightShoe`, the very first action may be either `LeftSock` or `RightSock`. A *total-order* plan is one that fully specifies the sequence in which the actions must occur; each total-order plan is a *linearization* of the partial-order plan. The partial-order solution shown in Figure 2.8 corresponds to the six total-order plans shown in Figure 2.9.

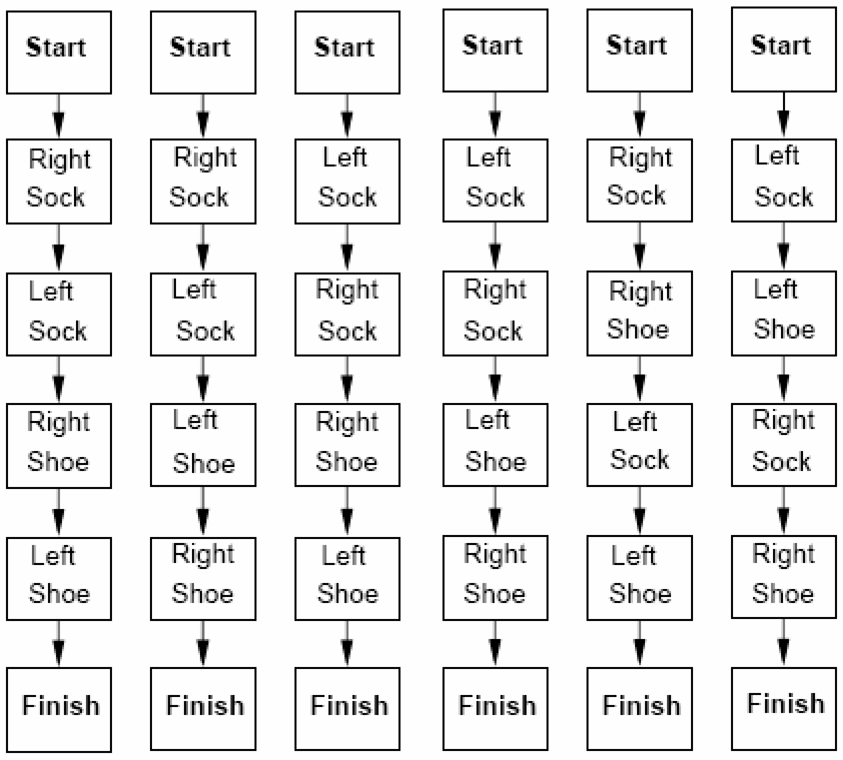


Figure 2.9: Six Corresponding Linearizations into Total-Order Plans

2.4.3 HTN Planning

The key idea underpinning Hierarchical Task Network (HTN) planning is that complexity is dealt with by hierarchical decomposition. Other examples of this approach include [AI Mod]:

- Software Development – created from a hierarchy of object classes
- Armies – operate as a hierarchy of units

In addition to an initial state, a goal state, and a set of operators, a HTN planning problem also includes a set of *methods*. Each method describes how a complex task can be decomposed into subtasks, and each operator describes how a primitive task can be achieved. HTN planning proceeds by using methods to decompose tasks recursively into simpler and simpler subtasks, until only primitive tasks remain in the plan. These primitive tasks should be directly achievable from the operators.

Figure 2.10 below describes a HTN house-building problem, involving seven operators (actions), and a single method `BuildHouse` (shown graphically).

```
GOAL:   (House)
INIT:   (Money ∧ GoodCredit)

ACTION:(BuyLand,     PRECOND: Money;      EFFECT: Land ∧ ¬Money)
ACTION:(GetLoan,     PRECOND: GoodCredit; EFFECT: Money ∧ Mortgage)
ACTION:(BuildHouse   PRECOND: Land;       EFFECT: House)
ACTION:(GetPermit    PRECOND: Land;       EFFECT: Permit)
ACTION:(HireBuilder                       EFFECT: Contract)
ACTION:(Construction PRECOND: Permit ∧ Contract;
                      EFFECT: HouseBuilt ∧ ¬Permit)
ACTION:(PayBuilder   PRECOND: Money ∧ HouseBuilt;
                      EFFECT: ¬Money ∧ House ∧ ¬Contract)
```

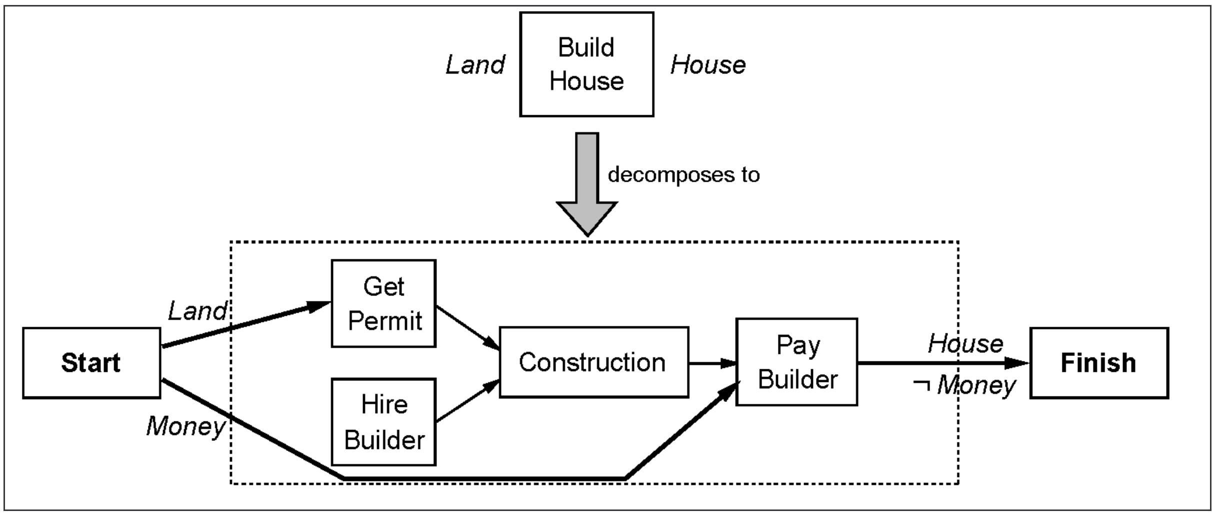


Figure 2.10: HTN House-Building Problem

Notice that the decomposition of `BuildHouse` is itself a plan. When BuildHouse is decomposed an additional precondition (`Money`) and effect (`¬Money`) are introduced which are not stated in the high-level description. In general, the high-level preconditions and effects are guaranteed to be a subset of the preconditions and effects of the decomposition [AI Mod].

The solution to this problem is shown is Figure 2.11. First of all, `BuyLand` is added to satisfy the `Land` precondition of `BuildHouse`. When `BuildHouse` is decomposed, this introduces an extra precondition `Money`, which is satisfied by adding `GetLoan`[4]

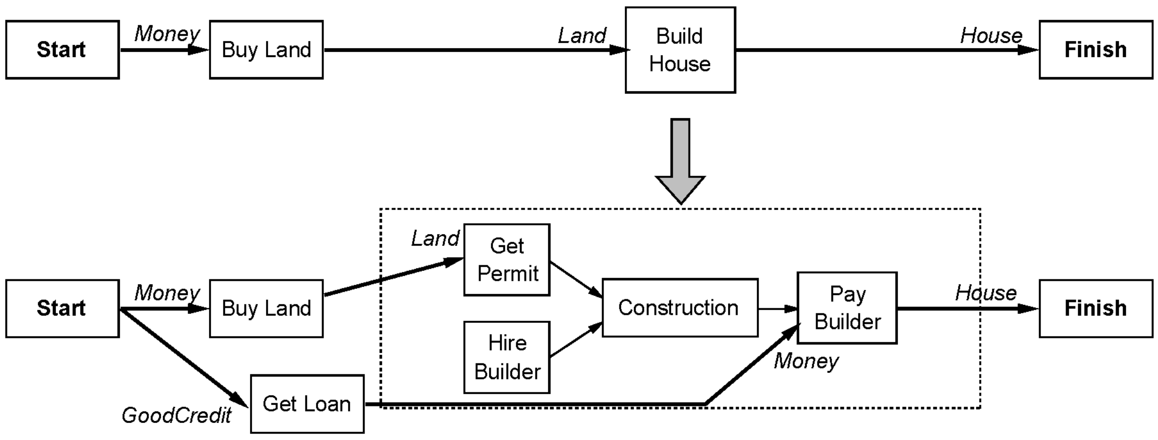


Figure 2.11: Solution to HTN House-Building Problem

[4] This solution is in fact a hybrid of partial-order planning and HTN planning, as the problem is not solved by task decomposition alone (due to the addition of `BuyLand` and `GetLoan`).

2.4.4 Suitability of HTN Planning to Web Service Composition

The appropriateness of HTN planning to the problem of Web Service composition is suggested by the many operational similarities between OWL-S and HTN planning:

- OWL-S atomic processes correspond to primitive actions
- OWL-S composite processes correspond to complex actions
- Both OWL-S and HTN planning use hierarchical decomposition to cope with complexity. Specifically, OWL-S uses control constructs to decompose composite processes to subprocesses, and HTN planning uses methods to decompose complex actions to simpler subtasks
- When a composite process or complex action is decomposed, the parameters of the subprocesses or subtasks may be either internal or external. In Figure 2.5, the external inputs (`Currency`, `BookName`) resulting from the decomposition of `BookPriceProcess`, must be satisfied by the invoking user/agent, whereas the external preconditions resulting from the decomposition of `BuildHouse` in Figure 2.11 are satisfied by adding `BuyLand` and `GetLoan`.

In fact, such is the similarity between OWL-S and HTN planning that OWL-S constructs can be mapped directly to those of HTN planning, e.g. composite processes to methods, and atomic processes to operators. Once this mapping has been performed, given an objective (or goal) and an initial state, a HTN planner will find a sequence of Web Services which achieves this goal from the initial state (if one exists).

Other reasons why HTN Planning is suitable for Web Service composition are [HTN 04], [PLAN SWS]:

- HTN encourages modularity. For example, the author of a method need not know how its subtasks may be further decomposed. This modularity fits well with the loosely coupled nature of Web Services
- HTN scales well to large numbers of methods and operators.

- Some HTN planners support complex precondition reasoning and provide natural places for human intervention. The latter could be used to query the user or an agent for input during planning, or if the planner cannot decompose a task, it could delegate that task decomposition to the user or an agent.

3 DESIGN & IMPLEMENTATION

3.1 Requirements

The high-level objective is to convert OWL-S descriptions to a HTN planning domain (i.e. a collection of methods and operators), which will be used by a HTN planner to solve a Web Service composition problem. This introduces the following requirements:

1. A means of reading OWL-S descriptions (programmatically)

   Although OWL-S descriptions may be read with any XML parser, a more efficient approach is to use an API designed specifically for reading OWL or better still, OWL-S.

2. A HTN planner

   Because there are a number of proven, freely available HTN planners, writing one's own should be considered an option of last resort.

3. An appropriate means of representing the problem state

   Because the IOPEs of semantic Web Services are expressed in OWL, the state of a Web Service composition problem should be represented as an OWL KB. However, AI planners typically represent the state as a list of ground literals, and support only fairly limited reasoning capabilities. Therefore, a means of representing the problem state in a manner that is consistent with the semantics of OWL is necessary if a generic HTN planner is used. Furthermore, in order to realise preconditions (and postconditions), a means of expressing and evaluating conditional expressions – by querying the state – is required.

4. An OWL reasoner

An OWL reasoner is needed to support type substitution. For example, if an input of a particular type is required, it should be possible to provide instead an instance which is:

- A `subClassOf` this type
- An `equivalentClass` to this type
- An `equivalentClass` to a `subClassOf` this type

In order to realise this, an OWL reasoner is required which can determine the relationship between OWL types. This reasoner should be able to reason over OWL classes defined in different ontologies.

3.2 Design & Implementation Technologies

This section introduces the technologies involved in OWLS2JSHOP. Their function in the context of OWLS2JSHOP is described in the next section.

3.2.1 JESS

The Java Expert Shell System (JESS) is a rule-based programming environment written in Java [JESS]. The data in a rule-based system is stored in facts, and the logic is defined as a collection of rules.

3.2.1.1 JESS Facts

A fact in a JESS KB is analogous to a row in a database table and a fact's slots are analogous to a table's columns. The structure of a fact is defined by a named template which specifies the name and number of slots in facts of this type. For example, the following statement defines a template named `person` with slots for storing an individual's `name`, `age`, and `gender`:

```
(deftemplate person (slot name) (slot age) (slot gender))
```

3.2.1.2 JESS Rules

A JESS rule is similar to an `if...then` statement in a traditional programming language, in that it consists of a conditional expression, and a series of commands to execute when that conditional expression is satisfied. The conditional expression occurs on the left-hand-side (LHS) of a rule and is also known as the *antecedent*. The commands to execute occur on the right-hand-side (RHS) and are known as the *consequent*. For example, the following defines a JESS rule named `foo-rule` which prints out: `"Bobby is 24 and male"` if a corresponding fact exists:

```
(defrule foo-rule
(person (name Bobby) (age 24) (gender male))
=>
(printout t "Bobby is 24 and male"))
```

3.2.2 Jena

Jena is a Java framework for building Semantic Web applications [Jena]. This framework includes:

- An RDF API
- Reading and writing RDF in RDF/XML, N3 and N-Triples
- An OWL API
- In-memory and persistent storage
- RDQL – a query language for RDF

3.2.3 OWLJessKB

This is a Java tool which translates an OWL ontology into a collection of JESS facts [OWLJess]. The facts created by OWLJessKB are defined by the following template:

```
(deftemplate triple (slot predicate) (slot subject) (slot object))
```

So in short, OWLJessKB converts an OWL ontology to a collection of PSO triples, which are stored in JESS. OWLJessKB performs the inferencing necessary to ensure that *all* the PSO triples entailed by an OWL ontology are stored as facts in JESS. OWLJessKB is implemented in Java, and depends on the Jena API.

3.2.4 OWL-S API

A Java API for reading, writing, and executing OWL-S service descriptions [OWL-S API]. The OWL-S API also depends on the Jena API.

3.2.5 Pellet

Pellet is an open-source OWL reasoner [Pellet] from the developers of the OWL-S API. Pellet is implemented in Java and depends on the Jena API.

3.2.6 JSHOP

3.2.6.1 Introduction

A HTN planner is a tool which can solve an AI planning problem given a:

- Domain Description – a list of operators and methods
- Problem Description – an initial state and a goal

The planner used by OWLS2JSHOP is JSHOP, a member of the SHOP family of HTN planners. Some of the reasons why this particular planner was chosen are given in subsection 3.4.1.

3.2.6.2 JSHOP Formalism

This section describes the elements of a JSHOP domain and problem definition. Much of the material in this section, and the example in the next are adapted from [JSHOP] [SHOP2].

3.2.6.2.1 JSHOP Domain Definition

A JSHOP domain consists of operators, methods, and axioms. Examples and definitions of these constructs are provided in this subsection.

An operator has the following form:

```
(:operator (!h) (P) (D) (A) [c])
```

where:

- h, the operator's *head*, consists of its name and parameter list
- P, the operator's precondition list
- D, the operator's delete list (list of negative effects)
- A, the operator's add list (list of positive effects)
- c, the operator's cost. It is possible to assign a cost to each operator and search for the cheapest plan available. This parameter is optional and defaults to 1 if omitted. This feature was not used.

Figure 3.1 shows two examples of JSHOP operators

```
(:operator (!pickup ?a) () () ((have ?a)))
(:operator (!drop ?a) ((have ?a)) ((have ?a)) ())
```

Figure 3.1: Examples of JSHOP operators

The `pickup` operator takes a single parameter, `?a`. It has no preconditions, and when it is invoked with a value bound to `?a`, `(have ?a)` is added to the state. When `drop` is invoked with a value bound to `?a`, it will only execute if `(have ?a)` is in the current state. If so, `(have ?a)` is removed from the state.

A JSHOP method has the following form:

```
(:method (h) (C₁)(T₁) (C₂)(T₂)...(Cₖ)(Tₖ))
```

where:

- h, the method *head*, names the task which the method decomposes
- Each C_i is a precondition list
- Each T_i, a *tail* of the method, is a task list which specifies how h may be decomposed into subtasks

A method indicates that h can be performed in k different ways by performing all of the tasks in exactly one of the method's tails when that tail's precondition list is satisfied. The preconditions are considered in the given order, so a later precondition is considered *only* if all of the earlier preconditions are not satisfied. Figure 3.2 shows an example of a method.

```
(:method (swap ?x ?y)
      ((have ?x)) ((!drop ?x)(!pickup ?y))
      ((have ?y)) ((!drop ?y)(!pickup ?x))
)))
```

Figure 3.2: Example of a JSHOP Method

This method provides two alternatives for decomposing `(swap ?x ?y)`. If `(have ?x)` is in the current state, the task may be decomposed may invoking operator `(!drop ?x)` followed by `(!pickup ?y)`. Alternatively, `?x` and `?y` may be swapped if `(have ?y)` is in the state, by calling `(!drop ?y)` followed by `(!pickup ?x)`.

A JSHOP axiom has the following form:

```
(:- (a) ((E₁)) ((E₂)) ((E₃))...((Eₙ)))
```

The intended meaning of an axiom is that `a` is true if any `Eᵢ` is true. One possible use of these axioms is inferencing. Consider the following:

```
(:- (mammal ?x)((dog ?x))((cat ?x))((human ?x)))
```

The effect of this axiom is that if the current state contains the term (dog rover) and (cat felix), the terms (mammal rover) and (mammal felix) will also be added. In other words, JSHOP will infer from this axiom that rover and felix are mammals.

3.2.6.2.2 JSHOP Problem Definition

A JSHOP problem definition has the form:

$$\texttt{(defproblem p d (}a_1\ a_2 \ldots a_n\texttt{) T)}$$

- p, is the name given to the problem
- d, is the name of the domain whose operators, methods, and axioms will be used in attempting to solve the problem
- The initial state of the problem is defined by the atoms $a_1 \ldots a_n$
- T is the list of tasks which must be completed, i.e., the problem goal

3.2.6.3 JSHOP Example

Figure 3.3 shows an example of a JSHOP domain and problem definition involving the operators and method shown in Figures 3.1, 3.2.

```
(defdomain basic-example (
  (:operator (!pickup ?a) () () ((have ?a)))
  (:operator (!drop ?a) ((have ?a)) ((have ?a)) ())

  (:method (swap ?x ?y)
    ((have ?x))
    ((!drop ?x) (!pickup ?y))
    ((have ?y))
    ((!drop ?y) (!pickup ?x)))))

(defproblem problem1 basic-example
  ((have banjo)) ((swap banjo kiwi)))
```

Figure 3.3: Complete JSHOP Example

The solution found by JSHOP is given in Figure 3.4, which shows the output printed by JSHOP when asked to solve this problem[5].

```
Problem file parsed successfully
Solving Problem :problem1
Returning successfully from find-plan : No more tasks to plan
1 plans found.
********* PLANS *******
Plan # 1
 ( (!drop banjo  )  1.0 (!pickup kiwi  )  1.0  )
```

Figure 3.4: JSHOP Solution

The solution shows the primitive actions, and the sequence in which they must be executed in order to achieve the objective: `(swap banjo kiwi)`, given the initial state: `(have banjo)`. The numbers beside the actions are their cost, which all default to 1.0.

3.3 OWLS2JSHOP Architecture

Figure 3.5 shows the architecture of the OWLS2JSHOP Web Service composition system. A short description of each component and its function is provided below. The manner in which these components interact to solve a Web Service composition problem is described in section 3.5.

[5] The verbosity of the output is controlled by a parameter which ranges from 0 to 10 (max). In this example, the parameter was set to 2.

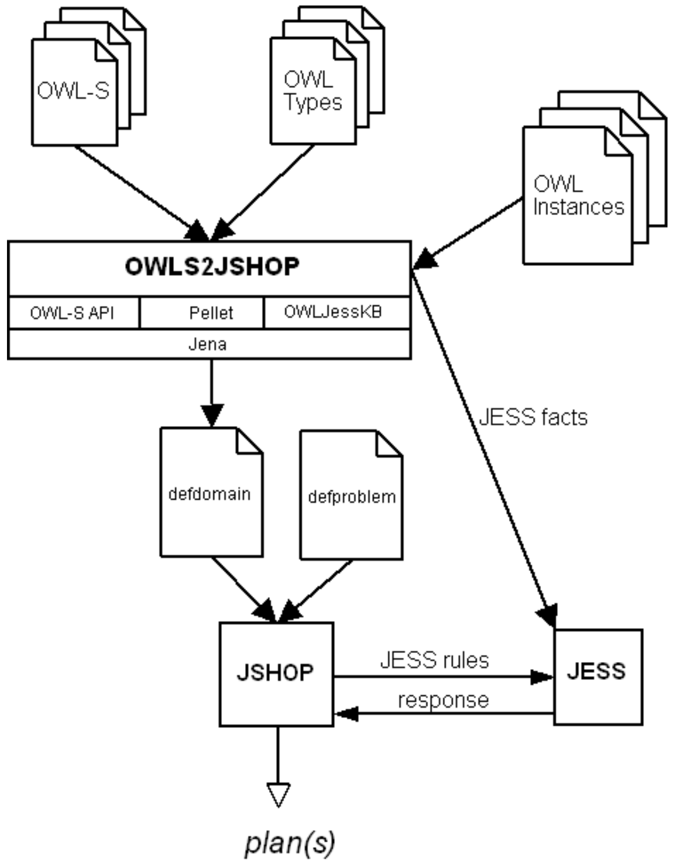


Figure 3.5: OWLS2JSHOP Architecture

- **OWL-S** – the OWL-S service descriptions that are translated by OWLS2JSHOP to a JSHOP domain definition file (**defdomain**).
- **OWL Instances** – the initial state of the planning problem is a set of OWL instances. These are read by OWLS2JSHOP, converted to a set of JESS facts and stored in JESS' KB
- **OWL Types** – OWLS2JSHOP supports type substitution, e.g. an output may be matched to an input if they are not of the same type, but the output type is a `subClassOf` or `equivalentClass` [OWL] of the input type. In order to achieve this, OWLS2JSHOP must be provided with the ontologies in which the input and output classes (and their subclasses and equivalent classes) are defined.
- **defproblem** – a JSHOP problem definition file in which the user specifies their goal (or objective) as a list of tasks to perform.
- **JSHOP** – the planner reads the domain and problem definitions, and attempts to find a plan. During the planning process, preconditions (and postconditions) are evaluated by querying JESS about the world state

- **OWLS2JSHOP** – the core component of the system which choreographs the Web Service composition process. OWLS2JSHOP is a Java package which interoperates with JSHOP and JESS via the Java APIs which these tools expose. It also depends on the following third-party libraries:
    - **OWL-S API** – a Java API for reading, writing, and executing OWL-S service descriptions [OWL-S API]
    - **Pellet** – an open-source Java based OWL DL reasoner [Pellet], which is used to discover the subclasses, superclasses, and equivalent classes of a given OWL class. This information is needed to enable type substitution.
    - **OWLJessKB** – this tool is used to translate the OWL instances that represent the initial state of the problem into a collection of JESS facts [OWLJess].
    - **Jena** – a Java API for building Semantic Web applications [Jena], which the OWL-S API, Pellet, and OWLJessKB all depend on

3.4 Design & Implementation Issues

3.4.1 Choosing the Planner

Choosing the correct planner is obviously critical to the success of any work involving AI planning. Reasons why JSHOP was chosen ahead of other HTN Planners include:

- It is implemented in the Java language, and therefore, is platform independent and can easily be extended or integrated with other Java applications, such as JESS and JSHOP.
- SHOP planners are widely used [SHOP '04] and available free-of-charge
- A SHOP planner won one of the top four prizes at the 2002 International Planning Competition [PLAN CO]
- SHOP planners have previously been used for Web Service Composition [HTN '04], [PLAN SWS]
- JSHOP provides advanced features such as: axiomatic inference, mixed symbolic/numeric computations, and calls to external programs [PLAN SWS]

Nevertheless, JSHOP is not without its flaws, specific disadvantages include:

- It is slower than some other SHOP planners, such as SHOP2. This is most likely attributable to the fact that SHOP2 is implemented in LISP. However this makes SHOP2 difficult to extend or integrate with Java applications
- The goal must be stated in terms of the tasks to perform, but in the context of Web Services a more natural way to express the objective is in terms of the required outputs and/or effects

3.4.2 State Representation

The IOPEs of semantic Web Services are expressed in OWL, and therefore the state of a Web Service composition problem should be represented as an OWL KB. However, JSHOP represents the state as a list of ground literals, and supports only fairly limited reasoning capabilities. Therefore it cannot cope with the expressivity of OWL. Before the problem of how to query the problem state (i.e. evaluate preconditions/postconditions) can be addressed, a means of representing the problem state in a manner that is consistent with the semantics of OWL is needed. One way of bridging the gap between an OWL KB and JSHOP's representation of the problem state, would be to replace the former with the latter. However, this would require drastic changes to the JSHOP source code, and in any event, something similar has already been achieved [PLAN SWS]. OWLS2JSHOP addresses this problem using JESS and OWLJessKB.

OWLJessKB converts the initial state – expressed as a set of OWL instances – to a collection of PSO triples, which are stored in JESS' KB. The semantics of any OWL ontology *can* be consistently represented as a collection of triples, indeed, this is exactly the representation used by arc-node diagrams (e.g. Figures 2.3, 2.4). Furthermore, OWLJessKB performs the inferencing necessary to ensure that *all* the PSO triples entailed by OWL instances representing the initial state are stored as facts in JESS.

In theory, it should be possible to keep the JESS KB up-to-date by adding/removing the appropriate OWL instances (translated to JESS facts) as JSHOP modifies the plan. For example, if JSHOP adds an operator that has an output which is an instance of the (fictitious) OWL class 'BookReceipt', JESS' KB could be kept up-to-date by translating this instance to JESS facts and adding them to the KB. However, OWLS2JSHOP doesn't actually invoke Web Services, and therefore this isn't possible. In other words, because we never actually execute the Web Service that produces the BookReceipt instance, we do not have a representation of it in OWL to translate. This problem doesn't arise when translating the initial state, because the OWL instances therein are provided externally, and not as the outcome of a Web Service execution.

3.4.3 Conditional Expressions

Preconditions are a key feature of AI planning, and can be defined as queries about the state which must be true in order for an action to be applicable. Similarly, postconditions – queries which must be true *after* an action – are of benefit to the planning process. For example, if an age-verifying service has a postcondition that the subject's age is at least 65, then this may be succeeded by a service which applies for a state old-age pension. In short, a postcondition provides a guarantee about the state after an action that the planner may use to determine which actions may succeed it.

Although neither OWL-S nor JSHOP support postconditions, a greater problem is that OWL-S 1.0 doesn't specify a language for encoding conditional expressions of any type. Clearly the issue of how to encode conditional expressions in OWL-S must be resolved before the matter of how to translate them to JSHOP constructs. Given that the state is represented as JESS facts, JESS rules are an obvious solution[6]. In fact, the decision to use JESS was motivated more strongly by this capability than the availability of a tool (OWLJessKB) to translate between OWL and JESS.
The LHS of a JESS rule which represents a precondition (or postcondition) will encode the terms of the condition in JESS, and the RHS will be empty as the consequent is implied, i.e. the action is applicable. For example, assume there exists

[6] JESS queries could also be used to encode conditional expressions

an OWL class Dog which has the URI http://www.example.org/Classes.owl#Dog. An instance of this class (defined in the same document as its class) such as: `<Dog rdf:ID="Rover"/>` would be represented in JESS' KB as the triple:

```
(triple
(subject     "http://www.example.org/Classes.owl#Rover")
(predicate   "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
(object      "http://www.example.org/Classes.owl#Dog"))
```

A process which has a precondition that an instance of this class must exist (in JESS' KB), could be expressed as the following JESS rule:

```
(defrule dog-precondition
(triple (subject ?x)
(predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
(object "http://www.example.org/Classes.owl#Dog"))
=>
)
```

Again, the RHS of this rule is empty because the issue of interest is whether the LHS is satisfied, that is, nothing needs to happen when it is. Rules such as this are evaluated by JESS which examines the KB and determines whether they can be activated. An activated rule is one whose LHS is satisfied and will be fired the next time the rules engine is run.

Encoding conditional expressions as antecedents of JESS rules is consistent with changes expected to be announced in OWL-S 1.1. This will specify the default language for encoding conditional expressions to be a variant of SWRL – essentially, SWRL rules without consequents – which is semantically equivalent to JESS rules without consequents. Although this SWRL variant will be specified as the default, any logical language such as DRS [DRS], Kif [KIF], or indeed JESS may be used.

On account of the fact that OWL-S doesn't support postconditions, extensions were made to the OWL-S ontology in order to provide support for postconditions in an OWL-S description. Also, because conditional expressions (of any sort) are currently unspecified in OWL-S, the source of the OWL-S API [OWL-S API] had to be modified in order to enable conditional expressions to be read (via the API) from an OWL-S description. These modifications make the process of translating an OWL-S precondition to a JSHOP precondition fairly straightforward, though the same cannot be said of postconditions, which JSHOP doesn't support. Rather than attempt to modify the source of JSHOP, the lack of explicit support for postconditions was circumvented thus: an atomic process `foo` with preconditions γ, postconditions δ, negative effects β, and positive effects α, this will be translated to the following JSHOP domain constructs:

```
(:method (foo) ((γ)) ((foo-op) (foo-pc)))
(:operator (!foo-op) ((γ)) ((β)) ((α)))
(:operator (!foo-pc) ((δ)) () ())
```

Less formally, an atomic process with postconditions is represented as a method with the same preconditions as the atomic process, which decomposes to a sequence of two operators. The first of these has the same preconditions and effects as the atomic process, and the second encodes the atomic process' postconditions as preconditions, and has no effects. This achieves the effect of postconditions, because whenever the atomic process `foo` is added to the plan, the net result will be the addition of `foo-op`, immediately followed by `foo-pc`. The former is identical to the atomic process in terms of its preconditions and effects, and the latter lists `foo`'s postconditions in its preconditions list. If any of these postconditions fail, both `foo-op` and `foo-pc` will be removed from the plan.

3.5 Component Interactions

Figure 3.6 shows a sequence diagram which describes how the components of the system combine to solve a Web Service composition problem. A description of the interactions between the components follows the diagram.

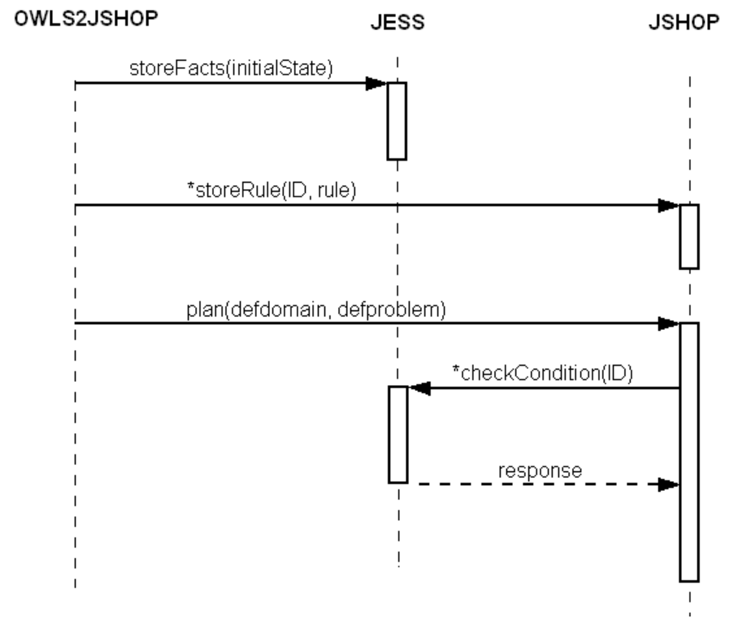


Figure 3.6: Sequence Diagram

- **OWLS2JSHOP** – recall that the inputs are the OWL-S descriptions themselves, the ontologies in which the classes of the inputs and outputs are defined, and the OWL instances which represent the initial state
  - `storeFacts(initialState)` – using OWLJessKB, the OWL instances which represent the initial state are converted to a set of JESS facts, which are stored in JESS' KB
  - `*storeRule(ID, rule)` – OWL-S preconditions and postconditions are expressed as JESS rules. However, JESS rules cannot be translated directly to JSHOP preconditions because the JSHOP parser prohibits various characters which may appear in rules. Instead, this operation stores the rules in a `Hashtable` within the JSHOP programme. The `Hashtable` key[7] is then used (instead of the rule itself), in the JSHOP precondition, which has the following format:

    (call JESS <*hashkey*>)
  - `plan(defdomain, defproblem)` – once the OWL-S description has been converted to a JSHOP domain definition file, it is passed to JSHOP along with the problem definition file, and JSHOP begins an attempt to find a plan

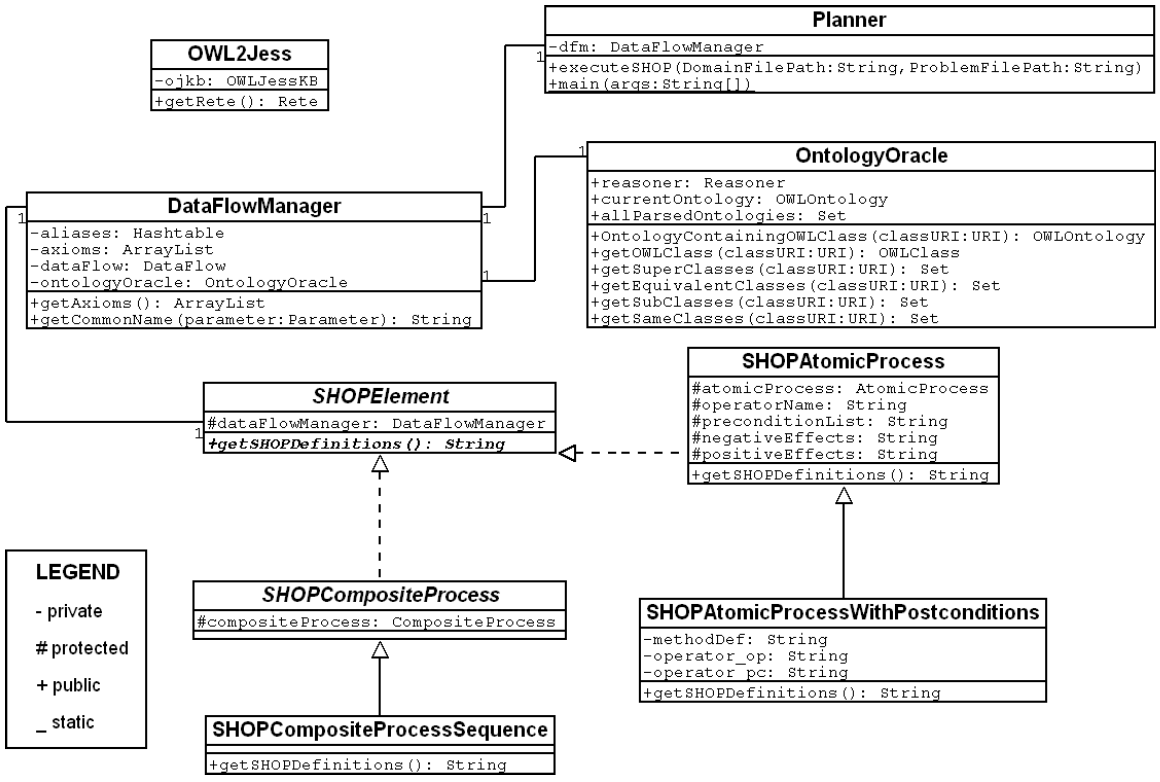
[7] The key is in fact the URI of the OWL-S precondition, with occurrences of '~' and '#' replaced by '¬', and '::' respectively. These substitutions are necessary because JSHOP prohibits the former two to be used within identifiers.

- **JSHOP, JESS** – once the domain and problem definition files have been passed to JSHOP, OWLS2JSHOP plays no further part. The process of solving the problem is the sole responsibility of JSHOP and JESS
    - checkCondition(ID) – when JSHOP encounters a precondition (or postcondition) of the form: `(call JESS <hashkey>)`, it is evaluated by retrieving the appropriate rule from the `Hashtable`, and passing it to JESS. If the rule is activated – an activated rule is one which will fire the next time the rules engine is run – JESS returns 'true', otherwise, it returns 'false'. Of course, the ability to interoperate between JESS and JSHOP using these preconditions is not a native feature of JSHOP, but was achieved via custom extensions to the JSHOP source code.

3.6 OWLS2JSHOP Package

This section describes the OWLS2JSHOP package. The source of this package along with the various libraries it depends on is included on the enclosed CD. Instructions for building OWLS2JSHOP are given in the `readme.txt` file.

Figure 3.7 shows a UML Class diagram which presents the core classes in this package and the relationships among them. Only attributes and public or protected methods (excluding constructors) are shown, and abstract classes or methods are displayed in italics.



Planner
-dfm: DataFlowManager
+executeSHOP(DomainFilePath:String,ProblemFilePath:String)
+main(args:String[])
OWL2Jess
-ojkb: OWLJessKB
+getRete(): Rete
OntologyOracle
+reasoner: Reasoner
+currentOntology: OWLOntology
+allParsedOntologies: Set
+OntologyContainingOWLClass(classURI:URI): OWLOntology
+getOWLClass(classURI:URI): OWLClass
+getSuperClasses(classURI:URI): Set
+getEquivalentClasses(classURI:URI): Set
+getSubClasses(classURI:URI): Set
+getSameClasses(classURI:URI): Set
DataFlowManager
-aliases: Hashtable
-axioms: ArrayList
-dataFlow: DataFlow
-ontologyOracle: OntologyOracle
+getAxioms(): ArrayList
+getCommonName(parameter:Parameter): String
1
1
1
1
1
1
SHOPElement
#dataFlowManager: DataFlowManager
+getSHOPDefinitions(): String
SHOPAtomicProcess
#atomicProcess: AtomicProcess
#operatorName: String
#preconditionList: String
#negativeEffects: String
#positiveEffects: String
+getSHOPDefinitions(): String
LEGEND
- private
# protected
+ public
_ static
SHOPCompositeProcess
#compositeProcess: CompositeProcess
SHOPAtomicProcessWithPostconditions
-methodDef: String
-operator_op: String
-operator_pc: String
+getSHOPDefinitions(): String
SHOPCompositeProcessSequence
+getSHOPDefinitions(): String

Figure 3.7: Core classes in OWLS2JSHOP package

OWLS2JSHOP also contains several classes which are not shown in Figure 3.7, as the functionality they provide is irrelevant to the main purpose of OWLS2JSHOP, i.e. converting an OWL-S `ServiceModel` description to a JSHOP domain description, and coordinating the other system components. These additional classes are described briefly below.

- Utilities – a collection of static methods, which are used by other classes in this package (e.g. `createFile`)
- MyAuthenticator – provides proxy server authentication
- Constants – provides a single point of reference for String constants used within this package (e.g. proxy server name)
- Exception subclasses

Additional information about the classes shown in Figure 3.7 is provided in the following subsections.

3.6.1 Planner

This is the main class of the system. An invocation of OWLS2JSHOP begins with the `main` method of this class which takes the following arguments:

1. URL of the top-level OWL-S document. This document may import other OWL-S documents (which may in turn import others) via the `owl:imports` construct
2. Proxy server username
3. Proxy server password
4. Directory location where the JSHOP domain file should be stored
5. Directory location of the JSHOP problem file. This argument is optional. If this argument is omitted a ready-made problem file will be used. This is a problem for which a solution was found during a previous invocation of OWLS2JSHOP with the same (1) value.

OWLS2JSHOP generates a JSHOP domain description file from an OWL-S ServiceModel description. Once this file has been created, `executeSHOP` is passed the location of the JSHOP problem and domain files. From this point onwards, execution is controlled exclusively by JSHOP and JESS.

3.6.2 OWL2JESS

This is a simple façade for OWLJessKB. The `getRete()` method returns a reference to the Rete object, which represents a running instance of JESS.

3.6.3 DataFlowManager

The data flow model maps inputs/outputs of different processes to one another (c.f. 2.2.3). For example, Figure 3.8 shows the OWL fragment which maps the `BookInfo` output of `BookFinderProcess`, defined in `BF.owl`, to the `BookInfo` input of `BNPriceProcess`, defined in `BNPrice.owl` (c.f. Figure 2.5).

```
<process:sameValues rdf:parseType="Collection">
 <process:ValueOf>
  <process:theParameter rdf:resource="http://www.example.org/BF.owl#BookInfo"/>
  <process:atProcess rdf:resource=" http://www.example.org/BF.owl#BookFinderProcess"/>
 </process:ValueOf>
 <process:ValueOf>
  <process:theParameter rdf:resource=" http://www.example.org/BNPrice.owl#BookInfo" />
  <process:atProcess rdf:resource=" http://www.example.org/BNPrice.owl#BNPriceProcess" />
 </process:ValueOf>
</process:sameValues>
```

Figure 3.8: OWL-S Parameter Mappings

Although the parameters mapped to one another in Figure 2.5 all have the same name, this is not required by OWL-S. However, in a JSHOP domain description, in order to refer to an entity in many places, the same identifier must be used in each. The purpose of DataFlowManager is to create a common alias for each set of parameters that are declared `sameValues` by the data flow model. These aliases ensure that parameters which are `sameValues` will use the same identifier in the JSHOP domain description.

3.6.4 OntologyOracle

This class is a façade for the Pellet `Reasoner` class. The `Reasoner` class provides methods to retrieve all the sub-classes, super-classes, equivalent classes of a given OWL class, but it can only search a single ontology at-a-time. OntologyOracle extends Pellet by providing similar methods which can reason over multiple ontologies. The `getSameClasses` method is used to implement type substitution. Given an OWL class, this method returns its equivalent classes, sub-classes, and the sub-classes' equivalent classes. The purpose of the other methods should be obvious from their signature.

3.6.5 SHOPElement

An abstract class that represents an element of a SHOP domain description (e.g. an operator or method). The `getSHOPDefinitions` method returns the text in the domain description file that defines this element to JSHOP.

3.6.6 SHOPAtomicProcess

This class represents an OWL-S atomic process. Atomic processes are represented in a SHOP domain description as an operator. Each of the attributes: `operatorName`, `preconditionList`, `negativeEffects`, and `positiveEffects` represent the corresponding components of a SHOP operator description.

3.6.7 SHOPAtomicProcessWithPostconditions

This class represents an OWL-S atomic process which has postconditions. These are represented in a SHOP domain description as a method which decomposes to two atomic processes. The method is represented by `methodDef`, and the atomic processes by `operator_op` and `operator_pc`.

3.6.8 SHOPCompositeProcess

This abstract class represents a generic OWL-S Composite process, i.e. the control construct that decomposes it is unspecified.

3.6.9 SHOPCompositeProcessSequence

This class represents an OWL-S Composite Process which is decomposed by the `Sequence` control construct. These composite processes are represented in a JSHOP domain description by a method.

4 SCENARIO

4.1 Introduction

This chapter describes in detail an OWLS2JSHOP use case. The scenario presented in this chapter involves all the features discussed previously: type substitution, preconditions, postconditions, etc. Instructions for building OWLS2JSHOP and running this scenario are provided in `readme.txt` on the enclosed CD.

Recall that the function of OWLS2JSHOP is to convert an OWL-S description to a JSHOP domain description, and coordinate the other system components. OWLS2JSHOP requires three inputs:

- OWL-S description of the service(s) which are to be converted to JSHOP
- OWL type hierarchy in which the classes of the inputs and outputs, and their subclasses and equivalent classes are defined
- OWL instances which define the initial state

4.2 Scenario Inputs

In this example, the top-level OWL-S document describes a service `BookPriceService` which has been adapted from [Book]. The OWL-S description of this service in XML format is provided in appendix A. The top-level process of `BookPriceService` is a composite process `BookPriceProcess` which decomposes to a sequence of three atomic processes: `BookFinderProcess`, `BNPriceProcess`, `CurrencyConverterProcess`. The names and types of the inputs and outputs of these processes are described in Figure 4.1 below. The composite process, `BookPriceProcess` takes the name of a book and an Asian currency, and returns the price of the book in this currency. This is achieved in three steps by the subprocesses, which perform the following transformations:

- `BookFinderProcess` – converts the book title to a `SportsBook` instance
- `BNPriceProcess` – returns the price of this book
- `CurrencyConverterProcess` – converts the price to the required currency

Parameters in Figure 4.1 which are declared `sameValues` by the data flow model are shown with the same line style (solid, dotted, etc.). For example, the output parameter of `BookFinderProcess` named `BookInfo` is mapped to the input parameter of `BNPriceProcess` with the same name.

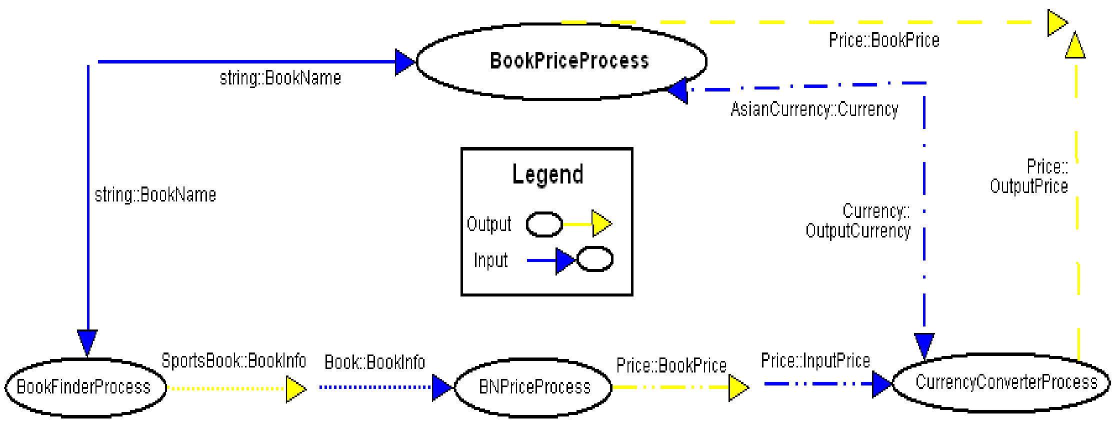


Figure 4.1: Type::Name of ServiceModel Inputs and Outputs

The type hierarchy of the input/output classes is shown in Figure 4.2[8]. Notice that in Figure 4.1 parameters which are sameValues do not necessarily have the same name e.g. `Price::BookPrice` and `Price::InputPrice`. Others have the same name, but different types, e.g. `SportsBook::BookInfo` and `Book::BookInfo`. In accordance with the usual rules of type substitution, two mapped parameters I and J may have different types if:

- I is an output of one process in a sequence, J is an input to the next process in the sequence, and I is a subclass of J, an equivalent class to J, or an equivalent class to a subclass of J, e.g. `SportsBook::BookInfo` and `Book::BookInfo`. If on the other hand J is a specialisation of I, the ServiceModel is invalid. For example, if the output of `BookFinderProcess` is a `Book` and the input to `BNPriceProcess` is a `SportsBook`, the ServiceModel is invalid because a `Book` is not a type of `SportsBook`

[8] Thing is the common base class of all OWL classes, like Object in Java

- I is an external output of an atomic process, J is an external output of a composite process and I is a subclass of J, an equivalent class to J, or an equivalent class to a subclass of J. Under these circumstances, the *actual* output of the service (which is that of the atomic process) will be a specialisation of the advertised output (which is that of the composite process).
- I is an external input to an atomic process, J is an external input to a composite process and J is a subclass of I, an equivalent class to I, or an equivalent class to a subclass of I, e.g. `AsianCurrency::Currency` and `Currency::OutputCurrency`. If however, I is a specialisation of J, the ServiceModel is invalid.

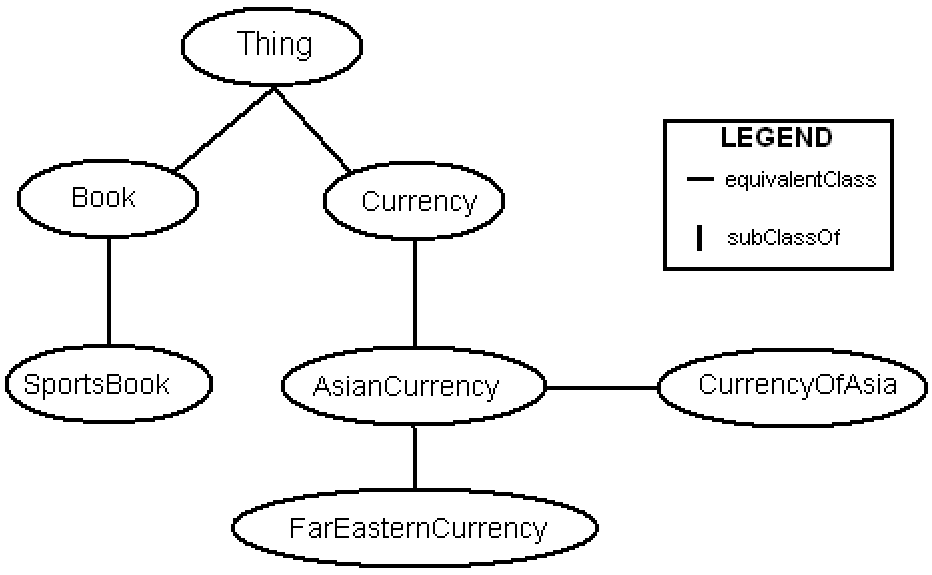


Figure 4.2: Type Hierarchy of Input and Output Classes

The third input required by OWLS2JSHOP is the initial state, expressed as a set of OWL instances. The initial state must satisfy the external inputs advertised by the top-level process, which in this case are an XSD string (`string::BookName`), and an instance of the `AsianCurrency` class (`AsianCurrency::Currency`). The latter has the following properties:

- code – the currency code (e.g. USD, GBP)
- name – the name of the currency
- country – country in which currency is legal tender

In order to demonstrate the type substitution feature, the initial state actually consisted of an XSD string and an instance of `FarEasternCurrency` – a subclass of `AsianCurrency`. The relevant OWL fragments are shown in Figure 4.3

```
<BookName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Ulysses
</BookName>

<paramTypes:FarEasternCurrency rdf:ID="KPW">
  <currency:code>KPW</currency:code>
  <currency:name>Won</currency:name>
  <currency:country rdf:resource="http://www.daml.org/2001/09/countries/iso#KP"/>
</paramTypes:FarEasternCurrency>
```

Figure 4.3: Scenario Initial State

In addition to the inputs and outputs shown in Figure 4.1, `CurrencyConverterProcess` also has a precondition and a postcondition. The meaning of the precondition is that the `currency:code` property value must be exactly three characters in length, and the postcondition guarantees that the length of this property value will be less than four characters. It should be obvious from Figure 4.3 that these conditional expressions should evaluate to 'true'. These conditional expressions are encoded in OWL-S as the antecedent of a JESS rule as shown in Figure 4.4.

```
<process:hasPrecondition>
<expression:Condition rdf:ID="PreValidCurrencyCode">
<expression:expressionLanguage rdf:resource="http://www.daml.org/services/owl-
s/1.1/generic/Expression.owl#JESS" />
 <expression:expressionBody>
(triple
(subject ?curr) (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type") (object
"http://www.cs.tcd.ie/~domurtag/owl-s/ParameterTypes.owl#FarEasternCurrency"))

(triple (subject ?curr) (predicate "http://www.daml.ecs.soton.ac.uk/ont/currency.owl#code")
(object ?literal))
```

```
(triple (subject ?literal) (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#value")
(object ?value &: (eq 3 (str-length ?value))))
</expression:expressionBody>
</expression:Condition>
</process:hasPrecondition>

<process2:hasPostcondition>
<expression:Condition rdf:ID="PostValidCurrencyCode">
 <expression:expressionLanguage rdf:resource="http://www.daml.org/services/owl-
s/1.1/generic/Expression.owl#JESS" />
<expression:expressionBody>
(triple (subject ?curr) (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
(object "http://www.cs.tcd.ie/~domurtag/owl-s/ParameterTypes.owl#FarEasternCurrency"))

(triple (subject ?curr) (predicate "http://www.daml.ecs.soton.ac.uk/ont/currency.owl#code")
(object ?literal))

(triple (subject ?literal) (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#value")
(object ?value &: (> 4 (str-length ?value))))
</expression:expressionBody>
</expression:Condition>
</process2:hasPostcondition>
```

Figure 4.4: Conditional Expressions Encoded as Conjunctions of JESS Atoms in OWL-S

Notice that the `hasPostcondition` element belongs to the `process2` namespace. This is the namespace which extends the OWL-S 1.0 ontology to provide support for postconditions. Details of these extensions are provided in XML format in appendix B.

4.3 Scenario Domain Definition

Given the inputs described in the previous section, OWLS2JSHOP produces the domain definition shown in Figure 4.5. Line numbers have been added to aid the explanation

```
(defdomain http://www.cs.tcd.ie/¬domurtag/owl-s/BookPrice.owl (

(:method
(BookPriceProcess)
((string:BookName:BookName ?x3)(AsianCurrency:Currency ?x1))
((!BookFinderProcess)(!BNPriceProcess)(CurrencyConverterProcess))
)

(:operator
(!BookFinderProcess)
((string:BookName:BookName ?x4))
()
((SportsBook:BookInfo))
)

(:operator
(!BNPriceProcess)
((Book:BookInfo))
()
((Price:BookPrice:InputPrice))
)

(:method
(CurrencyConverterProcess)
((Currency:OutputCurrency?x2)(Price:BookPrice:InputPrice)
(call JESS http://www.cs.tcd.ie/¬domurtag/owl-s/currencyconverter.owl::prevalidcurrencycode))
((!CurrencyConverterProcess-op)(!CurrencyConverterProcess-pc))
)

(:operator
(!CurrencyConverterProcess-op)
((Currency:OutputCurrency ?x2)(Price:BookPrice:InputPrice)
(call JESS http://www.cs.tcd.ie/¬domurtag/owl-s/currencyconverter.owl::prevalidcurrencycode))
()
((Price:OutputPrice:BookPrice))
)

(:operator
(!CurrencyConverterProcess-pc)
((call JESS http://www.cs.tcd.ie/¬domurtag/owl-s/currencyconverter.owl::postvalidcurrencycode))
()
()
)

(:-(AsianCurrency:Currency ?x)((FarEasternCurrency:Currency ?x))((CurrencyOfAsia:Currency ?x)))
(:-(Currency:OutputCurrency ?x)((AsianCurrency:Currency ?x)))
(:-(Book:BookInfo)((SportsBook:BookInfo)))))
```

Figure 4.5: Scenario Domain Definition

Each composite process is translated to a method (lines 3-7), each atomic process without postconditions is translated to an operator (lines 9-21), and each atomic process with postconditions is translated to a method which decomposes to two operators in the manner described in 4.2.3 (lines 23-43). The name of the domain is the URL of the top-level OWL-S description, with any occurrences of '~' replaced by '¬', and any occurrences of '#' replaced by '::', because the JSHOP parser doesn't permit these characters.

For example, the composite process `BookPriceProcess` is translated to a method of the same name, which decomposes to a sequence of three tasks (line 6) representing the subprocesses of the composite process. The inputs to `BookPriceProcess` are translated to the preconditions `(string:BookName:BookName ?x3)` and `(AsianCurrency:Currency ?x1)`. The variables in these preconditions `?x3` and `?x1` are placeholders for the values that must be provided for these external inputs. Although OWL-S makes a distinction between informational preconditions (inputs) and real-world preconditions (preconditions), JSHOP does not, so both OWL-S preconditions and inputs are represented as preconditions in JSHOP. Similarly, JSHOP does not distinguish between informational effects (outputs) and real-world effects (effects), so OWL-S outputs and effects are both represented as effects in JSHOP.

In order to refer to the same entity in many places in JSHOP, the same identifier must be used in each. Thus, parameters which are defined `sameValues` by the data flow model must be identified by the same name in JSHOP. If the parameters have the same type, this is achieved by using the alias assigned to this set of parameters by the DataFlowManager (c.f. 4.4.3). For example, the output `Price::BookPrice` and the input `Price::InputPrice` are mapped to each other by the data flow model, and are of the same type. The JSHOP alias is formed by taking the type of the parameters and appending the parameter names, i.e. `Price:BookPrice:InputPrice` (lines 20, 25, 32). If the parameters mapped by the data flow model have different (but compatible) types, then the parameter names and a JSHOP axiom are needed to indicate equivalency. For example, the output `SportsBook::BookInfo` and the input `Book::BookInfo` are mapped to each other by the data flow model, and the output is a subclass of the input. Equivalency is indicated by naming the former `(SportsBook:BookInfo)`, the latter `(Book:BookInfo)`, and the axiom (line 47):

```
(:-(Book:BookInfo)((SportsBook:BookInfo)))
```

Recall that the meaning of this axiom is that `(SportsBook:BookInfo)` infers `(Book:BookInfo)`. Therefore, the effect `(SportsBook:BookInfo)` will satisfy the precondition `(Book:BookInfo)` despite the fact that their identifiers differ. The axiom shown in line 46 has a similar intent; it ensures that the `AsianCurrency` external input of the composite process `BookPriceProcess` satisfies the `Currency` external input of the atomic process `CurrencyConverterProcess`. The other axiom (line 45) has a slightly different purpose. It is constructed by discovering all the subclasses, equivalent classes, and subclasses' equivalent classes of the external input `AsianCurrency:Currency` advertised by the top-level process – an equivalent axiom is not constructed for the other external input because its type is an XSD string rather than an OWL class. The effect of this axiom is that a specialisation of `AsianCurrency` may be provided instead of an `AsianCurrency`, behaviour which is equivalent to the OO concept of polymorphic substitution. The benefit of this behaviour is that it increases the number of services which may be invoked for a given problem state, thereby providing a wider range of service composition possibilities. For example, without this axiom it would not be possible to invoke the `BookPriceProcess` with a string instance and an instance of `FarEasternCurrency`. This feature of OWLS2JSHOP could not have been fully realised without extending Pellet to reason over multiple ontologies, because the specialisations of a particular class, will generally be defined in several disparate ontologies.

The OWL-S preconditions are encoded as JSHOP preconditions of the form: `(call JESS <rule_URI>)`, where `<rule_URI>` is the URI of the precondition, after substituting those characters prohibited by the JSHOP parser. When these preconditions are evaluated by JSHOP, an attempt to activate the corresponding rule is made – an activated rule is one which will fire the next time the rules engine is run – if the rule activates, the precondition evaluates to true, if not, it evaluates to false. Atomic processes with postconditions are represented as a method with the same name as the process (lines 23-28), which decomposes to two operators. The first of these (lines 30-36) is named by appending –op to the method name, and encodes the preconditions and effects of the atomic process, the second (lines 38-43) is named by appending –pc to the method name, and encodes the postconditions of the atomic process as operator preconditions.

4.4 Scenario Problem Description

The previous section describes the JSHOP domain produced by the planner. In order to produce a plan, a problem description is also required. The problem file used in this scenario is shown in Figure 4.6 below.

```
1    (defproblem my_prob http://www.cs.tcd.ie/¬domurtag/owl-s/BookPrice.owl
2    ((FarEasternCurrency:Currency KPW)(string:BookName:BookName Ulysses))
3    ((BookPriceProcess)))
```

Figure 4.6: Scenario Problem Definition

Line 3 describes the task list that represents the objective. In this case, the goal is simply to discover the sequence of primitive tasks that correspond to the top-level composite process, `BookPriceProcess`. The initial state is described by line 2, which represents an instance of the `FarEasternCurrency` class, and a string. It is worth emphasising that the initial state is represented within JSHOP simply as these two ground[9] literals:

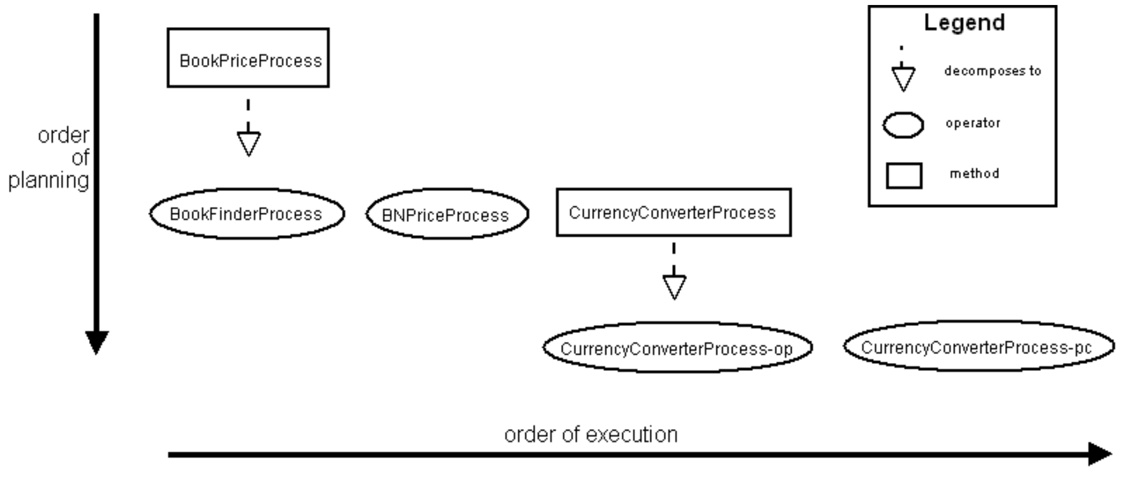
`(FarEasternCurrency:Currency KPW)` and `(string:BookName:BookName Ulysses)`

Although type substitution is simulated via axiomatic inferencing, JSHOP doesn't actually know anything about the FarEasternCurrency class, its properties, subclasses, superclasses, etc. This limitation, specifically, the inability to represent the semantics of OWL within JSHOP is the reason why JESS is employed by OWLS2JSHOP.

[9] The term 'ground' refers to the fact that all variables must be bound. The planner satisfies the preconditions of `BookPriceProcess` by binding `KPW` to `?x1` and `Ulysses` to `?x2`

4.5 Scenario Solution

It should be obvious that the problem described in Figure 4.6 is achievable by the domain described in Figure 4.5. The initial state satisfies the preconditions of `BookPriceProcess` and `BookFinderProcess` – the latter being the first primitive action in the decomposition of `BookPriceProcess`. The effects added by `BookFinderProcess` lead to the preconditions of `BNPriceProcess` being satisfied; whose effects in turn satisfy the preconditions of the method `CurrencyConverterProcess`. The latter is decomposed to `CurrencyConverterProcess-op` and `CurrencyConverterProcess-pc`, both of which are successfully added to the plan. The formation of the plan is represented graphically in Figure 4.7 below, along with the solution as printed by JSHOP. Recall that the plan printed by JSHOP shows the primitive actions, and the sequence in which they must be executed in order to achieve the objective. The cost of each action in the scenario solution defaults to 1.0.



```
1 plans found.
********* PLANS *******

Plan # 1
( (!bookfinderprocess ) 1.0 (!bnpriceprocess ) 1.0
(!currencyconverterprocess-op ) 1.0 (!currencyconverterprocess-pc )
1.0 )
```

Figure 4.7: Scenario Solution

5 RELATED WORK

5.1 Partial-Order Planning

One of the earliest attempts at applying AI planning to Web Service composition was presented by Sheshagiri et. al. in [PLAN '03]. The solution presented converts OWL-S `ServiceModel` descriptions into Predicate-Subject-Object (PSO) triples, which are used to build planning operators corresponding to each of the atomic services. The planner then chains backwards from the ultimate goal by repeatedly applying the following two steps:

1. Find operators that satisfy the existing goal(s) and add them to the plan
2. Convert the inputs and preconditions of these operators into a new set of goals

The planner terminates when it can no longer find any operators to satisfy the outstanding inputs and preconditions, which must then be provided externally. However this approach takes no account of those preconditions and inputs which are *actually capable* of being provided externally, and at worst, could find a plan which cannot be executed when one exists. This problem was avoided in OWLS2JSHOP by providing the planner with an explicit description of the initial state.

There are a number of similarities between OWLS2JSHOP and [PLAN '03]. For example, both build planning operators from OWL-S/DAML-S `ServiceModel` descriptions, and use JESS [JESS] to represent the problem state. However, the decision by the authors to write their own partial-order planner (using JESS) is questionable, given the plethora of high-quality, proven planners that are freely available. Their planner matches inputs to outputs and preconditions to effects on the basis of lexical matching of names alone, so it is conceivable that an output/effect with an inappropriate type could be chosen to satisfy an input/precondition, simply because they have the same name. However, OWLS2JHSOP considers the OWL type during matching, and supports type substitution, e.g. an output may be matched to an input if they are not of the same type, but the output type is a `subClassOf` or `equivalentClass` [OWL] of the input type. Finally, the authors' treatment of conditional outputs and effects is questionable; they assert that:

"It is not necessary for a composer to represent the control structure for conditional effects and outputs; merely enumerating them as multiple effects or outputs is sufficient for composition"

Although execution is beyond the scope of this system (and OWLS2JSHOP), this assumption could result in the failure of a plan at execution time, as the outputs/effects produced may differ from those assumed during planning.

[Mith '04] details enhancements to this partial-order planning system which include the addition of an execution engine. Unlike the HTN planning system described in [HTN '04], planning and execution are not interleaved, but instead, the entire plan is formed and then executed. In the event of plan execution failure, the execution engine can determine the cause of the failure, and ask the planner to build an alternative plan that doesn't involve the service responsible.

5.2 Planning and Execution

Although the Web Service composition solution discussed in the previous section is based on AI planning, it does not employ HTN planning, and therefore is somewhat different in approach to our own. However, the system developed by Sirin, Parsia, et. al. [HTN '04] [PLAN SWS] [INF] [AUT] does employ HTN Planning and therefore can readily be compared with ours. This system also uses a SHOP planner, albeit SHOP2, rather than JSHOP.

A distinguishing feature of this system is the interleaving of planning and execution. Specifically, services which have outputs (information-providing services) are executed during planning, meaning that the only services which appear in the final plan are those with effects. In order to achieve this, an assumption is made that services have either outputs or effects but not both [AUT]. The reason for this assumption is that services which are executed during planning may be invoked many times while the planner tries to find a plan. However, there is an obvious need to be cautious about executing services which have real-world effects, so if information-providing services are to be executed during planning, they must not also have effects.

Although this assumption is obviously untrue of many Web Services, it can be relaxed if the effects of the information-providing service do not interact with the plan being sought [INF]. For example, if an information-providing service charges a fee for its use, but the plan being sought has nothing to do with money, then this service can be safely executed during planning.

Reasons why it is desirable to execute an information-providing service during planning are described below:

- If the outputs of the service are conditional and it is executed during planning, the *actual* outputs will be known by the planner, thus enabling it to proceed with greater certainty and reducing the possibility of plan execution failing.
- If the service is unavailable, this will be discovered during planning, and an attempt can be made to discover an alternative plan which does not involve the faulty service.

Nevertheless, the point is made in [Mith '04] that it is unlikely to be often possible to execute information-providing services during planning because:

"Most services (except the services that constitute the head of the plan) are dependent on the execution of other services and cannot be executed in an arbitrary order. This is because these services need inputs that are available after the execution of other services"

Another argument against the interleaving of planning and execution is that service execution is a time-intensive process, and it is more efficient to only execute those services which are absolutely necessary, i.e. those that feature in the final plan. Although OWLS2JSHOP doesn't include an execution engine, it assumes there is no interleaving of planning and execution. Therefore, the plans produced by OWLS2JSHOP could be executed by an engine such as that described in [Mith '04].

5.3 Planning with Incomplete Information

An important difference between traditional AI planning, and AI planning for Web Service composition is discussed in [INF]. Traditional planning assumes that complete information about the initial state is available. However, as there are very many Web Services which can provide information about the state of the world, and it is infeasible to execute them all, the initial state in a Web Services composition problem is necessarily incomplete. The proposed solution to this problem is WSC-SHOP2, an algorithm for solving planning problems with incomplete information about the initial state. WSC-SHOP2 issues queries to learn the truth values of certain atoms when there is not enough information in the knowledge base to determine their values. Due to the unreliability of Web Services, while it is waiting for answers to these queries it does not cease planning, but searches for alternative plans that do not depend on the answers to the queries pending. This approach differs from the typical AI approach to planning with incomplete information, which is to insert information-gathering actions into the plan at the places where more information is needed. These information-gathering actions are executed during plan-execution, and depending on the information obtained, the planner chooses a pre-planned course of action. In other words, the typical AI approach defers information gathering until plan-execution whereas WSC-SHOP2 gathers information while planning. OWLS2JHSOP makes a 'closed world' assumption, meaning that any information about the state of the world must be provided upfront before planning begins. This approach is typical of more traditional applications of AI planning.

5.4 State Representation

Ideally, the state of a Web Service composition problem should be represented in OWL, and preconditions and effects expressed via OWL statements similar to SWRL atoms[10] [SWRL]. In order to use an AI planner to solve such a problem, it would have to understand the semantics of OWL, but planners typically support only fairly limited

[10] As mentioned in 2.2.3, OWL-S 1.0 does not mandate any language for encoding preconditions and effects, but it is expected that OWL-S 1.1 will make the default language for encoding preconditions and effect a variant of SWRL

reasoning capabilities (c.f. 2.4.2.1), and cannot cope with the expressivity of OWL ontologies. In order to bridge this gap between AI planners and OWL, an OWL reasoner was integrated with a SHOP2-based Web Service composition system [PLAN SWS]. Integration of an OWL reasoner with SHOP2 means that the planner's interaction with the problem state is delegated to the reasoner:

- The problem state itself is represented by the reasoner as an OWL knowledge base (KB)
- Evaluation of preconditions, which are written in OWL, is handled by the reasoner
- When service with effects written in OWL are added to the plan, updates to the state are performed by the reasoner

The intent of this work is very similar to our own, although the technologies involved differ. In order to represent state in a manner that preserves the semantics of OWL, an OWL KB is used in [PLAN SWS], whereas we use a JESS KB and OWLJessKB to translate from OWL to JESS. Similarly, we use JESS' rules syntax and engine to express and evaluate queries against this state, whereas [PLAN SWS] use OWL and an OWL reasoner.

5.5 Complex Action Planning

[COMP] describes an approach to Web Service composition that uses complex actions as the building blocks of a plan. These complex actions differ from those of HTN planning in that they are composed of primitive actions using traditional programming language constructs. For example, the complex action `goToAirprt(loc)` is defined as:

```
if loc=Univ
then shuttle(Univ, PA); train (PA, MB); shuttle (MB, SFO)
else taxi (loc, SFO)
```

The preconditions and effects of these complex actions can be precompiled under a frame assumption, which enables the complex actions to be treated as planning operators. Standard planning techniques (rather than HTN planning) are used to build a plan using these operators. This results in a plan in terms of complex actions, from which a plan in terms of primitive actions can be obtained if desired. It is claimed that this approach to planning can dramatically improve the efficiency of plan generation by reducing the size of the search space and the length of a plan.

5.6 Composing Web Services without Semantic Descriptions

All the systems discussed in this chapter so far are based on some form of AI planning and semantic markup of Web Services. However, there have been many attempts at Web Service composition that do not involve either one or both of these. However, systems which are based on WSDL descriptions alone (i.e. do not require semantic annotations), are either manual or only partially automated. Examples of such systems are BPEL4WS [BPEL4WS] and the Web Services Toolkit developed at IBM Research Laboratories [WSTK]. Although the latter uses a planner for service composition, construction of the operators from the service description is not fully automated because of the absence of a mechanism for capturing domain knowledge in WSDL.

A final example of a partially-automated Web Service composition system is [SWORD]. This system does not require services to be described in either DAML-S/OWL-S or WSDL. Instead, an entity-relationship model – which must be manually populated – is used to describe each atomic service's data inputs/outputs and conditional inputs/outputs11, and a JESS rule defines which outputs can be obtained from a service, given which inputs. In order to create a composite service, the user specifies the inputs and outputs of the composite service and submits it to SWORD, which uses JESS' rule engine to determine if the composite service can be realised. SWORD includes an execution model as well as a composition model, but it can only compose web services that do not have real-world effects. Although the SWORD method of composing services may seem radically different to ours, in fact, they are very similar, as an AI planner and a rules engine such as JESS are closely related. An example of a planner implemented in JESS has already been encountered [PLAN

[11] Conditional inputs/outputs in SWORD are equivalent to preconditions/effects in OWL-S

‘03], and, in one sense JESS itself is a planner, as it implements a superset of the PLANNER language [PLANNER], and thus basically does planning by simulation.

6 CONCLUSIONS

6.1 Evaluation

The objectives outlined in section 1.2 were:

1. Determine the most appropriate type of AI planning with regard to Web Service composition
2. Determine an effective means of representing the world/problem state
3. Identify a means of expressing conditional statements
4. Identify a means of evaluating conditional statements
5. Demonstrate the solution

A discussion on how these issues are addressed by OWLS2JSHOP follows.

6.1.1 AI Planning Type

HTN planning is an extremely good fit to Web Service composition, because hierarchical modelling is the key to both. The correspondence between the two is such that it is possible to directly map OWL-S concepts to those of HTN planning (e.g. composite processes to complex actions, and atomic processes to primitive actions), which means translating an OWL-S `ServiceModel` description to a HTN domain description is fairly straightforward. The appropriateness of HTN planning to Web Service composition has been exploited by other researchers [HTN '04] [PLAN SWS]; the motivation for employing this type of planning is discussed in detail in subsection 2.4.4.

6.1.2 State Representation

The HTN planning in OWLS2JSHOP is provided by the JSHOP planner. The reasons why this planner was chosen are outlined in subsection 3.4.1. Although JSHOP is one of the best HTN planners available, using it to compose Web Services presents a number of challenges. The most significant problem with JSHOP is its inability to represent the state of a Web Service composition problem as an OWL KB. It is this issue which is responsible for much of the complexity in OWLS2JSHOP. [PLAN SWS] overcomes this problem by integrating an OWL reasoner with a SHOP planner; OWLS2JSHOP addresses it by converting OWL to PSO triples, and storing them in JESS. A HTN planning tool which uses an OWL KB to represent state would negate the need for JESS (or an OWL reasoner), but such a tool is not available currently.

6.1.3 Conditional Expressions

Although conditions pervade OWL-S, the latest release does not specify any language for expressing them. It is expected that the next release of OWL-S will make a variant of SWRL the default language for expressing conditions, but the details have not yet been finalised, and is it clear how to evaluate such expressions. OWLS2JSHOP overcomes this problem by using conjunctions of JESS atoms to express conditions, which are evaluated by the JESS rules engine. The latter was achieved by extending the source code of JSHOP to support interoperation between JSHOP and JESS. Expressing conditional expressions as conjunctions of JESS atoms is consistent with the changes expected in the next OWL-S release, which will permit the use of any logical language for encoding conditional expressions.

6.1.4 OWLS2JSHOP Evaluation

OWLS2JSHOP demonstrates the appropriateness of applying HTN planning to Web Service composition, and highlights some of the key challenges involved. However, this system could be improved in several ways. Perhaps the most obvious improvement which could be made is the provision of support for a wider range of control constructs.

Although OWLS2JSHOP only provides explicit support for composite processes which are decomposed via the `Sequence` control construct[12], supporting control constructs which involve conditions (e.g. `If-Then-Else, Repeat-Until`) is fairly

[12] The `SHOPCompositeProcessSequence` class (Figure 4.3) could also support the Unordered control construct, which allows the subprocesses to be invoked in any order (including in sequence)

straightforward now that conditions can be expressed and evaluated. For example, in order to support the `If-Then-Else` control construct, an appropriate subclass of `SHOPCompositeProcess` (Figure 3.7) should be defined – assume it is named `ShopCompositeProcessIfThenElse`. If the subprocesses are atomic, this class should implement the method `getSHOPDefinitions`, such that the following SHOP elements are defined:

```
(:method (h) (C1)(T1) ()(T2))
(:operator (!T1) (P1) (D1) (A1))
(:operator (!T2) (P2) (D2) (A2))
```

where:

- h – h, the method *head*, names the composite process which it decomposes
- C_1 – the *if-condition* of the `If-Then-Else` control construct
- T_1 – the atomic process to execute if the if-condition is true
- T_2 – the atomic process to execute if the if-condition is false
- P_1, P_2 – precondition lists of T_1 and T_2 respectively
- D_1, D_2 – delete list of T_1 and T_2 respectively
- A_1, A_2 – add list of T_1 and T_2 respectively

If instead the subprocesses are composite, then T_1 and T_2 should be defined as methods rather than operators. In order to complete the support for composite processes that decompose via the `If-Then-Else` control construct, the code shown in bold below should be added to the private `processProcessList` method of the main class, `Planner`:

```
for (int i = 0; i < processes.size(); i++) {

        org.mindswap.owls.process.Process process =
                (org.mindswap.owls.process.Process)processes.get(i);

        if (process instanceof AtomicProcess) {
                AtomicProcess ap = (AtomicProcess)process;

                if (ap.getPostconditions().size() == 0) {
                        SHOPElements[i] = new SHOPAtomicProcess(ap, dfm);
```

```
			}
			else {
				SHOPElements[i] =
					new SHOPAtomicProcessWithPostconditions(ap, dfm);
			}
		}
		else if (process instanceof CompositeProcess) {
			CompositeProcess cp = (CompositeProcess)process;
			ControlConstruct cc = cp.getComposedOf();

			if (cc instanceof Sequence) {
				SHOPElements[i] = new SHOPCompositeProcessSequence(cp, dfm);
			}
			else if (cc instanceof IfThenElse) {
				SHOPElements[i] = new SHOPCompositeProcessIfThenElse(cp, dfm);
			}
		}
}
```

No changes are needed to call the `getSHOPDefinitions` method of the `SHOPCompositeProcessIfThenElse` objects as it will be called by the following lines of the private `writeDefDomain` method of `Planner`:

```
//write the 'defdomain' line
String fileContents = new String("(defdomain " + domainName + " (" + "\r\n");

//add the methods and operators
for (int i = 0; i < SHOPElements.length; i++) {
	fileContents += SHOPElements[i].getSHOPDefinitions();
}
```

A similar approach could be used to provide support for the other control constructs which depend on conditions (e.g. `Repeat-Until`, `Repeat-While`). Insufficient time is the main reason why support for these other control constructs wasn't actually implemented, but a lack of available OWL-S services that use these control constructs (which would be necessary for testing) is another factor. However, there are a couple of control constructs which cannot be supported by JSHOP, `Split` and `Split+Join`. The reason for this is that the components of these control constructs must be executed concurrently, which JSHOP does not support.

Another obvious improvement that could be made to OWLS2JSHOP is the addition of Web Service execution. The simplest way to achieve this would be to formulate the entire plan upfront, and pass it to an execution engine which invokes the relevant services via the OWL-S `ServiceGrounding`. A more complex solution is to interleave planning and execution. The pros and cons of interleaving planning and execution were discussed in detail in section 5.2.

6.1.5 Other Contributions

Another important contribution of OWLS2JSHOP is the provision of type substitution. This enables a wider range of service composition possibilities by permitting a specialisation of a class to be provided instead of an instance of the class itself. This was achieved using SHOP axioms, and required Pellet to be extended to enable reasoning over multiple ontologies.

Another novel feature of this work is the use of postconditions in service descriptions. A postcondition aids the planning process by providing a guarantee about the state after an action, which the planner may use to determine which actions may succeed it. Postconditions are expressed and evaluated in the same way as preconditions, but are not supported by OWL-S, so the OWL-S ontology was extended to enable postconditions to be included in an OWL-S description. Reading postconditions (and preconditions) from an OWL-S description required extensions to the OWL-S API.

6.2 Future Challenges

6.2.1 Service Discovery

In a broad context, the issue of service discovery will be crucial to the success of Web Service composition. This is concerned with how people will find Web Services suitable for service composition. Although this issue is outside the scope of OWLS2JSHOP, being able to find relevant Web Services in an efficient manner is obviously a prerequisite to composing them.

6.2.2 Scalability

A key issue to the success of an AI planning approach to composing Web Services is that of scale. Although it is claimed that HTN planning scales well [HTN 04], whether it can cope with composing Web Services at the scale of the Internet is unclear. Although discovering services and building the domain description could be performed offline (in much the same way as a search engine builds an index of keywords), solving a planning problem should be performed on-demand, though it may be possible to use caching to avoid repeatedly solving the same problem in quick succession.

6.2.3 OWL-S Issues

A number of other issues – many of which can be attributed to the recentness of OWL-S – have hindered the composition of Web Services on a large scale. The first of these is the paucity of OWL-S tools that are available. For example, it is only since the turn of the year (2004) that an OWL-S API has become available [OWL-S API]. Also, because any OWL-S tools that *are* available are invariably recently developed, they are often not as reliable as more mature tools. A second issue is the scarcity of OWL-S services themselves. A lack of OWL-S examples makes developing and testing OWL-S tools difficult, and creating your own OWL-S services can be a prohibitively lengthy process due to the verbosity of the XML format. Other difficulties with the semantic markup of Web Services include [EXP][13]:

- Lack of support for a service with multiple interfaces, that is, one which may accept different types of input parameters
- Conceptual model is imprecise. The many different models within OWL-S (`ServiceProfile`, `ServiceModel`, `ServiceGrounding`, WSDL), and the unclear links between them causes confusion and unnecessary complexity
- No clear correspondence between OWL-S and traditional software engineering concepts
- Mapping from OWL-S to WSDL limits expressivity

[13] The semantic Web Services markup referred to in this paper is in fact DAML-S, but the issues raised apply equally to OWL-S

However, perhaps the most important reason why OWL-S hasn't yet been widely deployed is its instability. In the last year alone, the *de facto* standard for marking up semantic Web Services has changed from DAML-S 0.9 to OWL-S 1.0, and a beta release of OWL-S 1.1 is currently available [OWL-S 1.1]. Extensive changes are proposed by the latter – particularly to the `ServiceModel` – examples include:

- A process' conditional outputs and effects are replaced by a `Result` class. A `Result` bundles together:
    - Zero or more effects
    - Zero or more outputs
    - The conditions under which these outputs and effects will occur.
- Preconditions, effects and results are logical expressions represented in SWRL, or another logical language such as DRS, or KIF.
- A new type of process parameter, `Local`, is defined. A `Local` parameter is a variable other than an input that is bound in a precondition of an atomic process.

Of course, every time the standard changes, corresponding changes must be made to dependent tools such as the OWL-S API. It is understandable that many would prefer to wait until the standard stabilises before developing any applications which depend on it.

Obviously the widespread adoption of OWL-S is crucial to the success of Web Service composition. There is little point in developing a sophisticated Web Service composition tool if the range of services available for composition is meagre. Fortunately, most of the issues mentioned above can be regarded as 'teething problems'. It is expected that as the standard stabilises, and the variety and quality of OWL-S tools available improves, OWL-S will be widely adopted as the semantic Web Service markup language of choice.

7 BIBLIOGRAPHY

[AI Mod] RUSSELL, S., NORVIG, P. 2003. *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice Hall. ISBN 0-130-80302-2

[AUT] WU, D., SIRIN, E., HENDLER, J., NAU, D., PARSIA, B. 2003. *Automatic Web Service Composition Using SHOP2*. Workshop on Planning for Web Services, Trento, Italy

[Book] *OWL-S Book Price Service* Available online at http://www.mindswap.org/2004/owl-s/1.0/BookPrice.owl

[Borst] BORST, W. N. 1997. *Construction of Engineering Ontologies for Knowledge Sharing and Reuse*. Ph.D. thesis, Universiteit Twente, The Netherlands

[BPEL4WS] *Business Process Execution Language for Web Services Version 1.1* Available online at: http://www-106.ibm.com/developerworks/library/ws-bpel/

[COMP] MCILRAITH, S., FADEL, R. 2002. *Planning with Complex Actions*. In Proceedings of the International Workshop on Non-Monotonic Reasoning

[Currency] *OWL Currency Ontology* Available online at: http://www.daml.ecs.soton.ac.uk/ont/currency.owl

[DRS] MCDERMOTT, D. 2004. *DRS: A Set of Conventions for Representing Logical Languages in RDF*. Available online at: http://www.daml.org/services/owl-s/1.0/drsguide.pdf

[EXP] SABOU, M., RICHARDS, D., VAN SPLUNTER, S. 2003. *An Experience Report on using DAML-S*. In Proceedings of the WWW03 Workshop on EServices and the Semantic Web, Budapest, Hungary

[HTN '04] SIRIN, E., PARSIA, B., WU, D., HENDLER, J., NAU, D. 2004. *HTN planning for web service composition using SHOP2*. In Journal of Web Semantics

[INF] KUTER, U., SIRIN, E., NAU, D., PARSIA, B., HENDLER J. 2004. *Information gathering during planning for web service composition*. In Submitted to Workshop on Planning and Scheduling for Web and Grid Services at ICAPS04

[Jena] *Jena Semantic Web Framework* Available online at http://jena.sourceforge.net/

[JESS] FRIEDMAN-HILL, E. 2003. *Jess in Action*, Manning. ISBN 1-930110-89-8

[JSHOP] YAMAN, F. 2002. *Documentation for JSHOP 1.0.1*. Dept. of Computer Science, University of Maryland. Available online at: http://www.cs.umd.edu/projects/shop/

[KIF] *Knowledge Interchange Format* Draft Proposed American National Standard Available online at: http://logic.stanford.edu/kif/dpans.html

[Mith '04] SHESHAGIRI, M. 2004. *Automatic Composition and Invocation of Semantic Web Services*. M.Sc. thesis, University of Maryland, USA

[OWL] *OWL Web Ontology Language Overview* Available online at: http://www.w3.org/TR/owl-features/

[OWL-S 1.0] *OWL-S 1.0 Release*
Available online at: http://www.daml.org/services/owl-s/1.0/

[OWL-S 1.1] OWL-S 1.1 Beta Release
Available online at: http://www.daml.org/services/owl-s/1.1B/

[OWL-S API] *OWL-S API*
Available online at http://www.mindswap.org/2004/owl-s/api/

[OWLJess] *OWLJessKB: A Semantic Web Reasoning Tool*
Available online at:
http://edge.cs.drexel.edu/assemblies/software/owljesskb/

[Pellet] *Pellet OWL Reasoner*
Available online at http://www.mindswap.org/2003/pellet/

[PLAN '03] SHESHAGIRI, M., DESJARDINS, M., FININ, T. 2003. *A Planner for Composing Services Described in DAML-S*. In AAMAS Workshop on Web Services and Agent-Based Engineering, Melbourne, Australia

[PLAN CO] *2002 International Planning Competition*
Available online at: http://planning.cis.strath.ac.uk/competition/

[PLANNER] HEWITT, C. E. 1969. *Planner: A language for Proving Theorems in Robots*. In Proceedings of the 1st IJCAI, Washington, D.C., USA

[PLAN SWS] SIRIN, E., PARSIA B. 2004. *Planning for semantic web services*. In Submitted to 3rd International Semantic Web Conference

[SHOP '04] NAU, D., AU, T. C., ILGHAMI, O., KUTER, U., MUNOZ-AVILA, H., MURDOCK, J. W., WU, D., YAMAN, F. 2004. *Applications of SHOP and SHOP2*. Available online at: http://www.cs.umd.edu/~nau/papers/nau04applications.pdf

[SHOP2] *Documentation for SHOP2*. Dept. of Computer Science, University of Maryland. Available online at: http://www.cs.umd.edu/projects/shop/

[SWORD] PONNEKANTI, S. R., FOX, A. 2002. *Sword: A developer toolkit for web service composition*. In Proceedings of the Eleventh World Wide Web Conference (Web Engineering Track), Honolulu, Hawaii, USA

[SWRL] HORROCKS, I., PATEL-SCHNEIDER, P. F., BOLEY, H., TABET, S., GROSOF, B., DEAN, M. 2004. *SWRL: A Semantic WebRule Language Combining OWL and RuleML*. W3C Member Submission. Available online at: http://www.w3.org/Submission/SWRL/

[WSDL] *WSDL Tutorial*
Available online at: http://www.w3schools.com/wsdl/default.asp

[WSDL 1.2] *WSDL Syntax*
Available online at: http://www.w3schools.com/wsdl/wsdl_syntax.asp

[WSTK] *alphaWorks: Web Services Toolkit, IBM*
Available online at:
http://www.alphaworks.ibm.com/tech/webservicestoolkit

8 APPENDICES

8.1 Appendix A

This appendix presents the OWL-S description of Figure 4.1 in XML format. Subsection 8.1.1 presents the description of the top-level composite process `BookPriceProcess`, and subsections 8.1.2-8.1.4 each describe one of the atomic subprocesses. The `ServiceGrounding` has been removed for the sake of brevity and the most relevant section of each description – the `ServiceModel` – is shown in bold. These descriptions are available in full in the owl-s directory of the enclosed CD, or online at: http://www.cs.tcd.ie/~domurtag/owl-s/

8.1.1 BookPriceProcess

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:owl=                "http://www.w3.org/2002/07/owl#"
  xmlns:rdfs=               "http://www.w3.org/2000/01/rdf-schema#"
  xmlns:rdf=                "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:factbook=           "http://www.daml.org/2003/09/factbook/factbook-ont#"
  xmlns:service=            "http://www.daml.org/services/owl-s/1.0/Service.owl#"
  xmlns:process=            "http://www.daml.org/services/owl-s/1.0/Process.owl#"
  xmlns:profile=            "http://www.daml.org/services/owl-s/1.0/Profile.owl#"
  xmlns:grounding=          "http://www.daml.org/services/owl-s/1.0/Grounding.owl#"
  xml:base=                 "http://www.cs.tcd.ie/~domurtag/owl-s/BookPrice.owl"
>

        <owl:Ontology rdf:about="">
                <owl:imports rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-
s/BookFinder.owl"/>
                <owl:imports rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-s/BNPrice.owl"/>
                <owl:imports rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-
s/CurrencyConverter.owl"/>
        </owl:Ontology>

<!-- Service description -->
<service:Service rdf:ID="BookPriceService">
        <service:presents rdf:resource="#BookPriceProfile"/>
        <service:describedBy rdf:resource="#BookPriceProcessModel"/>
</service:Service>

<!-- Profile description -->
<profile:Profile rdf:ID="BookPriceProfile">
        <service:isPresentedBy rdf:resource="#BookPriceService"/>
```

```
            <profile:serviceName xml:lang="en">Book Price Finder</profile:serviceName>
            <profile:textDescription xml:lang="en">Returns the price of a book in the desired currency.
First the ISBN number for the given book is found and then this ISBN number is used to get the prive
of the book from Barnes &amp; Nobles service.</profile:textDescription>

            <profile:hasInput rdf:resource="#BookName"/>
            <profile:hasInput rdf:resource="#Currency"/>
            <profile:hasOutput rdf:resource="#BookPrice"/>
</profile:Profile>

<!-- Process Model description -->
<process:ProcessModel rdf:ID="BookPriceProcessModel">
            <service:describes rdf:resource="#BookPriceService"/>
            <process:hasProcess rdf:resource="#BookPriceProcess"/>
</process:ProcessModel>

<process:CompositeProcess rdf:ID="BookPriceProcess">
            <process:hasInput rdf:resource="#BookName"/>
            <process:hasInput rdf:resource="#Currency"/>
            <process:hasOutput rdf:resource="#BookPrice"/>

            <process:composedOf>
                        <process:Sequence>
                                    <process:components rdf:parseType="Collection">
                                                <process:AtomicProcess
rdf:about="http://www.cs.tcd.ie/~domurtag/owl-s/BookFinder.owl#BookFinderProcess"/>
                                                <process:AtomicProcess
rdf:about="http://www.cs.tcd.ie/~domurtag/owl-s/BNPrice.owl#BNPriceProcess"/>
                                                <process:AtomicProcess
rdf:about="http://www.cs.tcd.ie/~domurtag/owl-
s/CurrencyConverter.owl#CurrencyConverterProcess"/>
                                    </process:components>
                        </process:Sequence>
            </process:composedOf>

            <process:sameValues rdf:parseType="Collection">
                        <process:ValueOf>
                                    <process:theParameter rdf:resource="#BookName"/>
                                    <process:atProcess rdf:resource="#BookPriceProcess"/>
                        </process:ValueOf>
                        <process:ValueOf>
                                    <process:theParameter
rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-s/BookFinder.owl#BookName"/>
                                    <process:atProcess rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-
s/BookFinder.owl#BookFinderProcess"/>
                        </process:ValueOf>
            </process:sameValues>

            <process:sameValues rdf:parseType="Collection">
                        <process:ValueOf>
                                    <process:theParameter
rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-s/BookFinder.owl#BookInfo"/>
                                    <process:atProcess rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-
s/BookFinder.owl#BookFinderProcess"/>
                        </process:ValueOf>
                        <process:ValueOf>
                                    <process:theParameter
rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-s/BNPrice.owl#BookInfo"/>
                                    <process:atProcess rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-
s/BNPrice.owl#BNPriceProcess"/>
```

```
				</process:ValueOf>
		</process:sameValues>

		<process:sameValues rdf:parseType="Collection">
				<process:ValueOf>
						<process:theParameter
rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-s/BNPrice.owl#BookPrice"/>
						<process:atProcess rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-
s/BNPrice.owl#BNPriceProcess"/>
				</process:ValueOf>
				<process:ValueOf>
						<process:theParameter
rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-s/CurrencyConverter.owl#InputPrice"/>
						<process:atProcess rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-
s/CurrencyConverter.owl#CurrencyConverterProcess"/>
				</process:ValueOf>
		</process:sameValues>

		<process:sameValues rdf:parseType="Collection">
				<process:ValueOf>
						<process:theParameter rdf:resource="#Currency"/>
						<process:atProcess rdf:resource="#BookPriceProcess"/>
				</process:ValueOf>
				<process:ValueOf>
						<process:theParameter
rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-
s/CurrencyConverter.owl#OutputCurrency"/>
						<process:atProcess rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-
s/CurrencyConverter.owl#CurrencyConverterProcess"/>
				</process:ValueOf>
		</process:sameValues>

		<process:sameValues rdf:parseType="Collection">
				<process:ValueOf>
						<process:theParameter
rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-s/CurrencyConverter.owl#OutputPrice"/>
						<process:atProcess rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-
s/CurrencyConverter.owl#CurrencyConverterProcess"/>
				</process:ValueOf>
				<process:ValueOf>
						<process:theParameter rdf:resource="#BookPrice"/>
						<process:atProcess rdf:resource="#BookPriceProcess"/>
				</process:ValueOf>
		</process:sameValues>
</process:CompositeProcess>


<process:Input rdf:ID="BookName">
		<process:parameterType
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
		<rdfs:label>Book Name</rdfs:label>
</process:Input>

<process:Input rdf:ID="Currency">
		<process:parameterType rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-
s/ParameterTypes.owl#AsianCurrency"/>
		<rdfs:label>Currency</rdfs:label>
</process:Input>

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

```
		<process:parameterType rdf:resource="http://www.mindswap.org/2004/owl-
s/concepts.owl#Price"/>
		<rdfs:label>Book Price</rdfs:label>
</process:Output>

</rdf:RDF>
```

8.1.2 BookFinderProcess

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:owl=		"http://www.w3.org/2002/07/owl#"
  xmlns:rdfs=		"http://www.w3.org/2000/01/rdf-schema#"
  xmlns:rdf=		"http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:service=	"http://www.daml.org/services/owl-s/1.0/Service.owl#"
  xmlns:process=	"http://www.daml.org/services/owl-s/1.0/Process.owl#"
  xmlns:profile=	"http://www.daml.org/services/owl-s/1.0/Profile.owl#"
  xmlns:grounding=	"http://www.daml.org/services/owl-s/1.0/Grounding.owl#"
  xml:base=		"http://www.cs.tcd.ie/~domurtag/owl-s/BookFinder.owl"
>

<owl:Ontology rdf:about="">
		<owl:imports
rdf:resource="http://www.mindswap.org/2003/owl/geo/geoCoordinateSystems.owl"/>
</owl:Ontology>

<!-- Service description -->
<service:Service rdf:ID="BookFinderService">
		<service:presents rdf:resource="#BookFinderProfile"/>
		<service:describedBy rdf:resource="#BookFinderProcessModel"/>
</service:Service>

<!-- Profile description -->
<profile:Profile rdf:ID="BookFinderProfile">
		<service:isPresentedBy rdf:resource="#BookFinderService"/>

		<profile:serviceName xml:lang="en">Book Finder</profile:serviceName>
		<profile:textDescription xml:lang="en">This service returns the information of a book whose
title best matches the give string.</profile:textDescription>

		<profile:hasInput rdf:resource="#BookName"/>
		<profile:hasOutput rdf:resource="#BookInfo"/>
</profile:Profile>

<!-- Process Model description -->
<process:ProcessModel rdf:ID="BookFinderProcessModel">
		<service:describes rdf:resource="#BookFinderService"/>
		<process:hasProcess rdf:resource="#BookFinderProcess"/>
</process:ProcessModel>

<process:AtomicProcess rdf:ID="BookFinderProcess">
		<process:hasInput rdf:resource="#BookName"/>
		<process:hasOutput rdf:resource="#BookInfo"/>
</process:AtomicProcess>

<process:Input rdf:ID="BookName">
```

```
            <process:parameterType
rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
            <rdfs:label>Book Name</rdfs:label>
</process:Input>

<process:Output rdf:ID="BookInfo">
            <process:parameterType rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-
s/ParameterTypes.owl#SportsBook"/>
            <rdfs:label>Book Info</rdfs:label>
</process:Output>
</rdf:RDF>
```

8.1.3 BNPriceProcess

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:owl=              "http://www.w3.org/2002/07/owl#"
  xmlns:rdfs=             "http://www.w3.org/2000/01/rdf-schema#"
  xmlns:rdf=              "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:service=          "http://www.daml.org/services/owl-s/1.0/Service.owl#"
  xmlns:process=          "http://www.daml.org/services/owl-s/1.0/Process.owl#"
  xmlns:profile=          "http://www.daml.org/services/owl-s/1.0/Profile.owl#"
  xmlns:grounding=        "http://www.daml.org/services/owl-s/1.0/Grounding.owl#"
  xml:base=               "http://www.cs.tcd.ie/~domurtag/owl-s/BNPrice.owl"
>

<owl:Ontology rdf:about="">
            <owl:imports
rdf:resource="http://www.mindswap.org/2003/owl/geo/geoCoordinateSystems.owl"/>
</owl:Ontology>

<!-- Service description -->
<service:Service rdf:ID="BNPriceService">
            <service:presents rdf:resource="#BNPriceProfile"/>
            <service:describedBy rdf:resource="#BNPriceProcessModel"/>
</service:Service>

<!-- Profile description -->
<profile:Profile rdf:ID="BNPriceProfile">
            <service:isPresentedBy rdf:resource="#BNPriceService"/>

            <profile:serviceName xml:lang="en">BN Price Check</profile:serviceName>
            <profile:textDescription xml:lang="en">This service returns the price of a book as advertised
in Barnes and Nobles web site given the ISBN Number.</profile:textDescription>

            <profile:hasInput rdf:resource="#BookInfo"/>

            <profile:hasOutput rdf:resource="#BookPrice"/>
</profile:Profile>

<!-- Process Model description -->
<process:ProcessModel rdf:ID="BNPriceProcessModel">
            <service:describes rdf:resource="#BNPriceService"/>
            <process:hasProcess rdf:resource="#BNPriceProcess"/>
</process:ProcessModel>

<process:AtomicProcess rdf:ID="BNPriceProcess">
            <process:hasInput rdf:resource="#BookInfo"/>
            <process:hasOutput rdf:resource="#BookPrice"/>
</process:AtomicProcess>

<process:Input rdf:ID="BookInfo">
            <process:parameterType rdf:resource="http://www.cs.tcd.ie/~domurtag/owl-
s/ParameterTypes.owl#Book"/>
            <rdfs:label>ISBN Number</rdfs:label>
</process:Input>

<process:Output rdf:ID="BookPrice">
            <process:parameterType rdf:resource="http://www.mindswap.org/2004/owl-
s/concepts.owl#Price"/>
            <rdfs:label>Book Price</rdfs:label>
</process:Output>

</rdf:RDF>
```

8.1.4 CurrencyConverterProcess

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE owl [
            <!ENTITY expr "http://www.daml.org/services/owl-s/1.1/generic/Expression.owl">
]>

<rdf:RDF
  xmlns:owl=                "http://www.w3.org/2002/07/owl#"
  xmlns:rdfs=               "http://www.w3.org/2000/01/rdf-schema#"
  xmlns:rdf=                "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:service=            "http://www.daml.org/services/owl-s/1.0/Service.owl#"
  xmlns:process=            "http://www.daml.org/services/owl-s/1.0/Process.owl#"
  xmlns:process2=           "http://www.cs.tcd.ie/~domurtag/owl-s/Process2.owl#"
  xmlns:profile=            "http://www.daml.org/services/owl-s/1.0/Profile.owl#"
  xmlns:grounding=          "http://www.daml.org/services/owl-s/1.0/Grounding.owl#"
  xmlns:expression=         "&expr;#"
  xml:base=                 "http://www.cs.tcd.ie/~domurtag/owl-s/CurrencyConverter.owl"
  >


<!-- This isn't part of the service description, just an OWL instance referred to therein -->
<expression:LogicLanguage rdf:ID="JESS">
            <expression:refURI
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">http://herzberg.ca.sandia.gov/jess/</ex
pression:refURI>
</expression:LogicLanguage>

<!-- Service description -->
<service:Service rdf:ID="CurrencyConverterService">
            <service:presents rdf:resource="#CurrencyConverterProfile"/>
            <service:describedBy rdf:resource="#CurrencyConverterProcessModel"/>
</service:Service>

<!-- Profile description -->
<profile:Profile rdf:ID="CurrencyConverterProfile">
            <service:isPresentedBy rdf:resource="#CurrencyConverterService"/>

            <profile:serviceName xml:lang="en">Price Converter</profile:serviceName>
		<profile:textDescription xml:lang="en">Converts the given price to another
currency.</profile:textDescription>

		<profile:hasInput rdf:resource="#InputPrice"/>
		<profile:hasInput rdf:resource="#OutputCurrency"/>

		<profile:hasOutput rdf:resource="#OutputPrice"/>
</profile:Profile>

<!-- Process Model description -->
<process:ProcessModel rdf:ID="CurrencyConverterProcessModel">
		<service:describes rdf:resource="#CurrencyConverterService"/>
		<process:hasProcess rdf:resource="#CurrencyConverterProcess"/>
</process:ProcessModel>

<process:AtomicProcess rdf:ID="CurrencyConverterProcess">
		<process:hasInput rdf:resource="#InputPrice"/>
		<process:hasInput rdf:resource="#OutputCurrency"/>
		<process:hasOutput rdf:resource="#OutputPrice"/>
		<process:hasPrecondition>
				<expression:Condition rdf:ID="PreValidCurrencyCode">
						<expression:expressionLanguage rdf:resource="&expr;#JESS"/>
						<expression:expressionBody>(triple (subject ?curr) (predicate
&quot;http://www.w3.org/1999/02/22-rdf-syntax-ns#type&quot;) (object
&quot;http://www.cs.tcd.ie/~domurtag/owl-s/ParameterTypes.owl#FarEasternCurrency&quot;))
(triple (subject ?curr) (predicate
&quot;http://www.daml.ecs.soton.ac.uk/ont/currency.owl#code&quot;) (object ?literal)) (triple
(subject ?literal) (predicate &quot;http://www.w3.org/1999/02/22-rdf-syntax-ns#value&quot;)
(object ?value &amp;: (eq 3 (str-length ?value))))</expression:expressionBody>
			</expression:Condition>
		</process:hasPrecondition>

		<process2:hasPostcondition>
		<expression:Condition rdf:ID="PostValidCurrencyCode">
						<expression:expressionLanguage rdf:resource="&expr;#JESS"/>
						<expression:expressionBody>(triple (subject ?curr) (predicate
&quot;http://www.w3.org/1999/02/22-rdf-syntax-ns#type&quot;) (object
&quot;http://www.cs.tcd.ie/~domurtag/owl-s/ParameterTypes.owl#FarEasternCurrency&quot;))
(triple (subject ?curr) (predicate
&quot;http://www.daml.ecs.soton.ac.uk/ont/currency.owl#code&quot;) (object ?literal)) (triple
(subject ?literal) (predicate &quot;http://www.w3.org/1999/02/22-rdf-syntax-ns#value&quot;)
(object ?value &amp;: (> 4 (str-length ?value))))</expression:expressionBody>
			</expression:Condition>
		</process2:hasPostcondition>

</process:AtomicProcess>
<process:Input rdf:ID="InputPrice">
		<process:parameterType rdf:resource="http://www.mindswap.org/2004/owl-
s/concepts.owl#Price"/>
		<rdfs:label>Input Price</rdfs:label>
</process:Input>

<process:Input rdf:ID="OutputCurrency">
		<process:parameterType
rdf:resource="http://www.daml.ecs.soton.ac.uk/ont/currency.owl#Currency"/>
		<rdfs:label>Output Currency</rdfs:label>
</process:Input>

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

```
            <process:parameterType rdf:resource="http://www.mindswap.org/2004/owl-
s/concepts.owl#Price"/>
            <rdfs:label>Output Price</rdfs:label>
</process:Output>

</rdf:RDF>
```

8.2 Appendix B

This appendix shows the extensions to the OWL-S 1.0 ontology (in XML format) which provide support for process postconditions. The relevant OWL file (process2.owl) is available in the owl-s directory of the enclosed CD, or online at: http://www.cs.tcd.ie/~domurtag/owl-s/Process2.owl

```
<?xml version='1.0' encoding='ISO-8859-1'?>

<!DOCTYPE uridef[
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
  <!ENTITY owl "http://www.w3.org/2002/07/owl">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY service "http://www.daml.org/services/owl-s/1.0/Service.owl">
  <!ENTITY process "http://www.daml.org/services/owl-s/1.0/Process.owl">
  <!ENTITY process2 "http://www.cs.tcd.ie/~domurtag/owl-s/Process2.owl">
  <!ENTITY DEFAULT "http://www.cs.tcd.ie/~domurtag/owl-s/Process2.owl">
]>

<rdf:RDF
  xmlns:rdf        ="&rdf;#"
  xmlns:rdfs       ="&rdfs;#"
  xmlns:owl        ="&owl;#"
  xmlns:xsd        ="&xsd;#"
  xmlns:service    ="&service;#"
  xmlns:process    ="&process;#"
  xmlns            ="&DEFAULT;#">

<owl:Ontology rdf:about="">
  <owl:imports rdf:resource="&service;"/>
</owl:Ontology>

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

<owl:ObjectProperty rdf:ID="postCondition">
  <rdfs:domain rdf:resource="#Postcondition"/>
  <rdfs:range rdf:resource="&process;#Condition"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasPostcondition">
  <rdfs:domain rdf:resource="&process;#Process"/>
  <rdfs:range rdf:resource="#Postcondition"/>
</owl:ObjectProperty>

</rdf:RDF>
```