TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Juha Puttonen

A Semantically Enhanced Approach for Orchestration of Web Services in Factory Automation Systems

Julkaisu 1224 • Publication 1224

Tampere 2014

Tampereen teknillinen yliopisto. Julkaisu 1224
Tampere University of Technology. Publication 1224

Juha Puttonen

A Semantically Enhanced Approach for Orchestration of Web Services in Factory Automation Systems

Thesis for the degree of Doctor of Science in Technology to be presented with due permission for public examination and criticism in Festia Building, Auditorium Pieni Sali 1, at Tampere University of Technology, on the 15th of August 2014, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2014

ISBN 978-952-15-3315-0 (printed)
ISBN 978-952-15-3339-6 (PDF)
ISSN 1459-2045

Puttonen, Juha: A Semantically Enhanced Approach for Orchestration of Web Services in Factory Automation Systems

Tampere University of Technology, Department of Mechanical Engineering and Industrial Systems, Finland, 2014

Keywords: SEMANTIC WEB SERVICES, SERVICE ORCHESTRATION, FACTORY AUTOMATION, WEB SERVICES, CLOUD COMPUTING

Abstract

The Service-oriented Architecture (SOA) paradigm makes it possible to build systems from several independent components. Most typically, web services are chosen as the building blocks of such a system. A web service is essentially a passive software entity, which listens for request messages sent to it over the network, possibly reacts to the requests by performing some operations, and finally sends response messages to the request senders.

The traditional application domain of web services belongs to the so-called IT domain. While opening new horizons in software development life-cycles, web services have been adopted in various new application domains, including the domain of factory automation (software development for factory automation). Indeed, recent research projects have experimented with controlling production system equipment through web service interfaces. When migrated from pure software to the physical realm involving industrial equipment, web services set additional demands for the application domains. For example, since the domains involve operations with physical effects, roll-back or application recovery procedures become challenging. This research work targets the orchestration of factory automation systems encapsulated as web services and presents various techniques for overcoming the difficulties.

Orchestrating web services to accomplish a complicated production task can be difficult due to the transitoriness of both production equipment states and the set of available web services. Nevertheless, the selection of appropriate web services can be facilitated by augmenting each service with semantic information describing its conditions and effects. Web services augmented with such descriptions are termed semantic web services. While Web Ontology Language, OWL, is ideal for describing application domain concepts and property relationships, the OWL-S ontology, which is based on OWL, has been specifically developed for describing web services. Once the semantic service descriptions have been analyzed to find the appropri-

ate web services, the selected services can be invoked using their syntactic WSDL descriptions.

In addition to automated web service selection, semantic descriptions allow the composition of web services to achieve production tasks. Service composition involves first analyzing the descriptions to determine the appropriate service invocation process for achieving the desired goal and then executing the process. This dissertation presents an approach in which the production equipment and their states are represented using an ontology, and the model is dynamically used in decision-making. In particular, the devices in the considered production systems provide web service interfaces through which they can be controlled, while semantic web service descriptions formulated in OWL-S make it possible to determine the conditions and effects of invoking the web services. The approach presented in this research work additionally involves a set of specialized web services that co-operate to achieve production goals using the domain web services. One of the services maintains a semantic model of the current system state, while another uses the model to compose the domain web services so that they jointly achieve the desired goals. The semantic model of the system is automatically updated based on event notifications sent by the domain services.

Software agents controlling production devices must maintain an up-to-date view of the physical world state in order to efficiently reason and plan their actions. Especially in a factory automation system, the world state undergoes rapid evolution, and the world view must remain synchronized with the changes. This research discusses two approaches to updating the world view based on event notifications sent by web services representing production devices in a manufacturing system. One of the approaches is based on separately specified update rules, and one automatically uses semantic web service descriptions formulated in OWL-S. While all of the examples presented in this research work specifically focus on the factory automation domain, the presented approaches are applicable to all domains involving semantic web services.

Semantic Web Service descriptions facilitate the automated discovery and composition of web services. Particularly in the production system domain, the service condition and effect descriptions are essential in selecting the appropriate service or service composition for a given task. OWL-S is one of the most popular semantic web service description languages, and due to its XML syntax, OWL-S can be effortlessly incorporated into service WSDL descriptions. However, developing OWL-S documents for each service instance is laborious. This dissertation presents an approach to automatically generating executable OWL-S descriptions from se-

mantically annotated service WSDL files.

Computing clouds facilitate rapid and effortless resource allocation. Cloud consumers can generally be ignorant of the physical computing resources used or their geographical location, as the resources are abstracted into a commodity that can be dynamically leased from the cloud provider. In particular, Infrastructure-as-a-Service clouds allow clients to dynamically lease virtual machines that behave similarly to physical servers. However, executing an application by directly using computing cloud resources is complicated and typically involves similar steps as installing and executing an application on a physical machine. Moreover, starting numerous application instances on a single virtual machine may result in poor performance. Thus, this dissertation considers the development of a web service that facilitates the use of cloud resources by abstracting them. When the web service is used, an application can be effortlessly started in a computing cloud by invoking simple web service operations. Furthermore, when multiple applications are started, the workload can be automatically distributed between several virtual machines, resulting in higher performance.

To conclude, the results presented in this research work demonstrate that semantic web service descriptions can indeed facilitate automatic web service composition and invocation. However, the effort of developing semantic web service descriptions can partly undermine the benefits achieved through their application. Therefore, new tools and methods should be developed to minimize the effort of developing such descriptions.

Acknowledgements

I would like to thank Professor José L. Martinez Lastra for supervising this research work and my doctoral studies. José also provided invaluable guidance in composing this dissertation.

I am very grateful to Dr. Andrei Lobov for supervising my research work. Andrei has provided the concepts for several of the software tools presented in this dissertation.

This research was made possible by all of the FAST Lab. members, who made the working environment comfortable as well as motivated me through their interest in the developed software tools and several useful improvement suggestions. Furthermore, the experiments with the production line devices were possible only because of the abundant support from my co-workers. In addition, my co-workers provided me descriptions of the production systems, and I have incorporated some of the images into the figures in Chapter 5.

Finally, I want to thank my parents and brother for all of the support and advice they have provided.

Pori, July 2014

Juha Puttonen

FOREWORD

The research presented in this dissertation was partly carried out in the context of the SOCRADES and eSONIA projects.
SOCRADES[1], contract number IST-5-034116, is part of the Information Society Technologies (IST) initiative of the 6th Framework Programme of the European Union.
eSONIA[2] was funded by ARTEMIS Joint Undertaking[3], and the project was carried out by a consortium of 14 partner organizations.

[1] http://www.socrades.eu/
[2] http://esonia.eu/
[3] http://www.artemis-ju.eu/

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
BFS	Breadth-First Search
BPEL	Business Process Execution Language
CPU	Central Processing Unit
DAML-S	The DARPA Agent Markup Language for Services
DFS	Depth-First Search
DL	Description Logic
DPWS	Devices Profile for Web Services
FIFO	First In, First Out
GUI	Graphical User Interface
IaaS	Infrastructure as a Service
IDE	Integrated Development Environment
MAS	Multi-Agent System
OOP	Object-Oriented Programming
OWL	Web Ontology Language
OWL-S	OWL for Service
PaaS	Platform as a Service
PDDL	Planning Domain Definition Language
PLC	Programmable Logic Controller
RAM	Random-Access Memory
RDF	Resource Description Framework
RTU	Remote Terminal Unit
SaaS	Software as a Service

SAWSDL	Semantic Annotations for WSDL and XML Schema
SOA	Service-Oriented Architecture
SQWRL	Semantic Query-enhanced Web Rule Language
SWRL	Semantic Web Rule Language
SWS	Semantic Web Service
UDDI	Universal Description Discovery and Integration
UML	Unified Modeling Language
UPnP	Universal Plug and Play
URI	Uniform Resource Identifier
WS	Web Service
WSDL	Web Services Description Language
WSMO	Web Service Modeling Ontology
WSRF	Web Services Resource Framework
XML	Extensible Markup Language
XPath	XML Path Language
XSL	Extensible Stylesheet Language Family
XSLT	XSL Transformations

1. INTRODUCTION

Factory automation systems are modern, automated manufacturing systems. The typical tasks of such a system include assembling of products from separate components. A major task in the development of a factory automation system is the control of the constituent devices, such as conveyors and assembly robots. Development of embedded devices has made it possible to embed miniature computers into the production equipment. In particular, the programmable logic controller (PLC) is universally applied in industry [126]. With the rapid increase in the computing capacity of the controllers, factory automation systems have become software-intensive.

1.1 Motivation and Justification

Software engineering methodologies emphasize the encapsulation and reuse of software components. Service-oriented architecture (SOA), on the other hand, makes it possible to encapsulate both software and hardware inside web service interfaces [35]. In manufacturing systems, such encapsulation facilitates the replacement of devices without additional reprogramming efforts [12].

While web services are the preferred implementation mechanism for an SOA, other technologies, such as UPnP (Universal Plug and Play), are also applicable [35]. However, this dissertation considers only SOAs implemented on web services based on the open web service specifications, such as WS-Discovery [68] and WS-Eventing [119], although several of them remain unstandardized. The main research target is the semantics-based orchestration of web services in factory automation systems. Therefore, web services are primarily considered to conform to the DPWS (Devices Profile for Web Services) [17] specification. DPWS specifies the minimal implementation requirements for web services compliant with a core set of web service specifications and is particularly intended for the device space; each web service is hosted on a device [35].

Orchestration of web services fundamentally encompasses two interrelated problems. Firstly, it is necessary to compose service execution plans. Secondly, the plan execution requires robust control methods.

To allow the remote discovery and control of production devices, web services embedded at device-level are a natural solution. Each web service exposes an interface containing different operations. Clients may invoke the operations by sending

certain types of messages to the service over the network.

To encapsulate production devices as web services, the device controllers must host the web service interfaces and mediate the service requests and responses. Some commercially available RTUs (Remote Terminal Units) already provide this capability.

The web service interfaces exposed by device controllers would typically contain operations corresponding to the actions that the production devices are expected to perform [36]. For example, the controller of a conveyor segment might expose a web service interface containing operations for transferring a pallet in and out of the segment. Encapsulating production equipment in web service interfaces enables the range of tools and methodologies developed for describing and orchestrating web services to be also applied in the factory automation domain. In particular, the methodologies for orchestrating web services make it possible to prescribe production processes without dependence on any vendor-specific languages [35]. In addition, the process descriptions are independent of the hardware used in the production system. If, for example, the controller devices were changed to a different model, it would be sufficient to ensure that the new controllers expose similar web service interfaces for all previous process flow descriptions to remain applicable. Thus, the approach makes process descriptions applicable to several different settings and reduces the complexity of production system device control.

The Service-oriented architecture (SOA) has been proposed for solving challenges, such as dynamically connecting new devices, in manufacturing systems [35]. In particular, when industrial devices are encapsulated and exposed as web services, they can be controlled using public web service standards [35]. Furthermore, it is straightforward to develop web service interfaces, even for most legacy systems [39].

For example, the Web Services Business Process Execution Language (WS-BPEL) [3] allows the description of complex work-flows involving several manufacturing devices. However, the direct application of BPEL in flexible manufacturing systems can be challenging due to rapid changes in the available services and service interfaces [81].

Unfortunately, traditional web service description languages, such as the Web Service Definition Language (WSDL) [8], convey only the information required to invoke the services. In particular, they define the syntax of service requests and responses. Effectively, the selection and composition of appropriate services for a given task requires a human expert. For example, a WSDL document typically lists the operations that the web service supports and describes the required input parameters and the produced output parameters. Thus, a WSDL document only describes the syntax of the information exchange with a web service, and further knowledge is required to deduce the meaning of the operations and the exchanged

data values.

As the traditional web service orchestration methodologies depend on syntactic web service interfaces, changing a web service interface syntactically by, for example, renaming an operation typically renders the service incompatible with existing process descriptions. Semantic web service descriptions provide a solution to this problem, as they describe the meaning of a web service and its effects. Hence, syntactic differences in web service interfaces become less relevant.

Semantic representation languages also make it possible to model web service states or the states of the devices represented by web services. An accurate model of the world state is vital for efficient orchestration of web services; orchestrating web services to co-operatively carry out a complex production process requires that sudden changes in the states of the production equipment and available web services are considered. For this purpose, this dissertation demonstrates the dynamic updating of semantic production system equipment and web service models.

Web services enriched with semantic descriptions are called semantic web services [62]. Semantic web service technologies facilitate the automation of web service discovery, invocation, and composition [62]. The efficient application of semantic web service technologies requires an automated framework. This dissertation focuses on using a set of web services that co-operate to successfully orchestrate domain web services.

For example, semantic annotations can facilitate the selection of matching web services for the partner links declared in a BPEL process [80]. However, the approach fails to exploit the full potential of semantic web service descriptions. In particular, changes to the set of available web services tend to invalidate the BPEL processes. Moreover, it is difficult to assess the semantic resemblance between two different condition or effect expressions.

Semantic web service descriptions facilitate the automatic composition of web services to achieve complex goals. Automatic service composition eliminates the need to describe work flows syntactically. For example, the production of a certain product can be accomplished by simply formulating an expression that describes the desired goal state. Query languages, such as SPARQL [79], allow queries to be formulated which, when evaluated over a semantic model of a production system, specify whether a production goal has been satisfied. A software agent can then use the queries and semantic web service descriptions for composing and executing process prescriptions that achieve the goals [89]. However, the use of semantic service descriptions enriched with condition and effect expressions in the achievement of complex goals while maintaining an accurate domain description still requires dedicated tools and methods, to which this research work contributes.

Computing processes require hardware resources, such as processing power and

data storage capacity. Traditionally, the resources have existed on physical server machines. Hence, organizations have had to purchase the hardware as well as allocate resources in installing and maintaining the systems. Moreover, the need for computing resources tends to considerably fluctuate, causing the expensive systems to be frequently idle. Adjusting the amount of computing resources to match the current needs is typically expensive using traditional methods. Cloud computing provides a solution to the problem by allowing organizations to lease computing resources and only pay for the amount that they actually use [110]. In addition, the cloud consumers can remain ignorant of service implementation details, such as the physical location of the computing resources employed.

1.2 Problem Statement

Although web service interfaces considerably facilitate the control of production devices, as such, they are insufficient to completely remove the difficulties in instructing a production system to perform the desired activities. In particular, when the web services are described using traditional, syntactic languages, it is laborious to formulate workflow prescriptions automating web service orchestration. Such prescriptions are vulnerable to minor changes in the syntactic web service interfaces. To automate the selection, invocation and composition of web services, semantic web service descriptions have been proposed [62].

Web services are software and inherently require hardware resources on which to run. A recent trend is to outsource computationally intensive tasks into computing clouds. However, the workload in deploying an application on cloud resources can still be considerable [93].

To summarize, the main questions that this research attempts to answer are:

- How to automate the development of semantic web service descriptions including the service pre- and post-conditions?
- Given a certain production goal, how to automatically compose and control the execution flow of the semantic web services that deliver the manufacturing system functionality?
- How to facilitate the effective use of cloud-based computing resources?

1.3 Research Objectives and Initial Hypotheses

The main objective is to develop methodologies allowing the development of more intelligent production systems. The production systems are able to reason on semantic information and achieve production goals more autonomously. This will

increase automation and eliminate the need to manually devise and modify detailed workflow prescriptions.

An additional objective is to facilitate the use of resources leased from computing clouds. In particular, a cloud consumer should be able to start applications on the leased resources without detailed knowledge on the individual resources.

The main research hypothesis is that the second research problem listed in Section 1.2 can be solved through the following three principles.

- The production goals are expressed in machine-interpretable form.
- The functionalities provided by the web services are semantically represented.
- The aforementioned two principles form the basis for the automatic composition and execution of machine-interpretable web service workflows.

An additional research hypothesis is that a web service deployed on a virtual machine in an Infrastructure-as-a-Service cloud can facilitate the use of the leased computing resources. The web service interface provides operations allowing the execution and termination of applications.

1.4 Contributions

This dissertation presents a semantic web service orchestration framework that is capable of automatically composing and executing web service workflows when the orchestrated services represent manufacturing equipment. In particular, the orchestration framework is implemented as a set of web services, and service composition can be initiated by submitting a production goal through one of the web service interfaces provided. Implementing the components of the orchestration framework as web services renders them more robust to implementation changes and allows them to be geographically dispersed as long as there is a network connection between the web services.

The orchestration framework maintains a record of all available semantic web services, and the AI planning process that forms the core of the service composition task uses the semantic service descriptions as its main input data. Additionally, the planning process input data includes a description of the current world state, which is stored and constantly updated in an OWL model hosted by one of the framework services.

The updating of the domain model is mainly based on event notifications received from the domain web services, which in the factory automation domain are production devices, such as robots and conveyors. As event notifications are automatically received, it is unnecessary to periodically inquire service statuses. Nonetheless,

periodic inquiry may be incorporated when the domain services provide no event notifications.

Furthermore, the framework can execute web service workflows expressed as BPEL processes. The BPEL processes may store production goals and submit them to the orchestration framework, or they can contain specific domain web service invocation work-flows.

As OWL-S descriptions and the embedded condition and effect expressions are typically stored in XML text files, they can be created and edited without any specialized tools. However, formulating a separate OWL-S document for each web service can be laborious. Therefore, this dissertation presents an approach of automatically generating OWL-S descriptions for semantic web services based on the service WSDL descriptions.

In addition, this dissertation presents a mechanism that assesses cloud resource utilization rates to optimize web service deployment. The approach reduces the amount of idle resources and increases the amount of resources available to the deployed services.

1.5 Thesis Outline

The remainder of this dissertation is structured as follows.

Chapter 2 will present some of the most notable prior achievements in the research field and briefly compare them to the new approaches presented in this dissertation. Chapter 3 will then present the new concepts and solutions in detail. Chapter 4 will discuss how the developed approaches and the related software tools have been implemented in practice. Chapter 5 will demonstrate the use of the tools in production system scenarios. Finally, Chapter 6 will summarize the contributions and identify issues to be addressed in further research.

2. STATE OF THE ART

This chapter reviews research that has been performed on web services, semantic web services, and cloud computing. First, Section 2.1 presents a short survey of graphical software tools facilitating the manual development of web service (WS) workflow prescriptions. Then, Section 2.2 shifts the focus from traditional WS to semantically described WS. Finally, Sections 2.3 and 2.4 briefly present recent advancements in Cloud Computing and agent-based systems, respectively.

A large portion of the research presented in this chapter has provided foundations for the new methods that form the topic of Chapter 3. The new methods aim to solve the deficiencies that Section 2.5 identifies in the current state of the art.

2.1 Graphical BPEL Tools

BPEL processes and the activities of which they are composed can be graphically visualized, and there are already several graphical BPEL editors available.

Graphical BPEL editors considerably reduce the burden on the developer, as writing XML code by hand is somewhat error-prone. Furthermore, generating BPEL processes with a graphical editor tends to result in much cleaner code [52]. In addition, observing the overall structure of a web service is typically easier from the graphical representation than the XML code.

2.1.1 NetBeans

NetBeans[1] is an IDE which can be used for several purposes. Some versions also include a graphical BPEL editor and a debugger. Web Services can be tested by creating and executing test cases. In addition, NetBeans is integrated with an application server for running the created web services.

For example, creating partner links with the graphical BPEL editor in NetBeans is effortless. The user must only drag the partner link element from the palette onto the canvas and select an existing partner link type. Alternatively, the user can create a new partner link type definition to a WSDL file in a wizard that pops up. Furthermore, NetBeans provides wide control over the created BPEL processes, as it is also possible to manually edit the BPEL process XML source code. Thus,

[1] Available at `https://netbeans.org/`. Accessed on 2014-6-19.

it is possible to, for example, copy-paste BPEL code into the editor and manually edit it to suit specific needs. Unfortunately, NetBeans supports only a subset of the WS-BPEL 2.0 specification. More specifically, it uses the Sun BPEL Service Engine (SE), which excludes some language constructs of WS-BPEL 2.0 [4]. These limitations may cause difficulties in importing BPEL processes created using other tools into NetBeans. Moreover, the graphical BPEL editor is somewhat inconvenient in that the user can only drag and drop BPEL elements into fixed locations in the canvas. In particular, NetBeans provides cursor hints to indicate the locations into which a BPEL element can be placed in the canvas. Nevertheless, the approach removes the need for separate location information, thereby facilitating the import of BPEL processes from other BPEL editors, since the graphical layout of BPEL processes is automatic.

Because NetBeans also supports Java development, web service processes can also incorporate services implemented as, for example, Java Beans projects. Java Beans projects can be deployed on the application server used by NetBeans and can therefore easily be used as part of more complex web services.

At the time of writing this dissertation, BPEL process support has been removed from the latest versions of NetBeans. Nonetheless, prior versions of NetBeans, including those with BPEL support, are still available.

2.1.2 ActiveBPEL

ActiveBPEL Designer[2] has been developed specifically for creating BPEL processes. The graphical user interface (GUI) is based on Eclipse. ActiveBPEL Designer can be downloaded free of charge. Although it is possible to view the XML source code created by the graphical BPEL editor, the source code cannot be manually edited. Effectively, the user is forced to use the GUI elements for all BPEL editing. This is a considerable disadvantage because an experienced user may find direct source code editing somewhat faster and easier than editing a BPEL process through the GUI components. Moreover, copy-pasting BPEL source code into the editor does not seem to be possible. On the other hand, the graphical editor is quite flexible, and the user can freely place BPEL elements on the canvas. Although the locations of the elements remain during saving and loading of projects, the location information is not stored in the XML code of the BPEL processes. Thus, the generated BPEL files contain completely valid BPEL code with no custom conventions, which might cause compatibility problems with other tools. However, if a BPEL process is imported into ActiveBPEL from some other tool, the location information is naturally missing, and the BPEL elements appear on top of each other in the canvas. Then, the user

[2]Available at http://www.activevos.com/. Accessed on 2008-4-30

must drag them to the appropriate locations to make the diagram comprehensible.

ActiveBPEL Designer allows the simulation of processes without deploying them on a server. Input values to the simulated BPEL process can be set before the simulation is started, and the values of variables and the current execution state can be observed at each step of the simulation.

The processes created using ActiveBPEL Designer can be deployed on an ActiveBPEL server. ActiveBPEL Designer includes an ActiveBPEL server, the ActiveBPEL Engine, which can be used in developing web services. In addition, the proprietary ActiveBPEL Enterprise can be used to directly deploy web services. ActiveBPEL Enterprise allows several versions of the same web service to run simultaneously.

When a new partner link type is created in ActiveBPEL, the user may specify a new or existing WSDL file in which the new partner link type will be stored. Unfortunately, partner link types refer to WSDL files to specify the port types used by the partner link roles, and ActiveBPEL Designer includes no editing support for WSDL files. Instead, WSDL files must be created using some other tool and then imported into ActiveBPEL projects as web references. On the other hand, a plugin allowing, for example, the graphical creation and editing of WSDL files is available for ActibeBPEL Designer [1].

2.1.3 Eclipse

Eclipse is a general-purpose integrated development environment (IDE) that can be extended to support various technologies through its built-in plugin mechanism. For example, graphical BPEL editing tools can be installed into the Eclipse IDE free of charge as a set of plugins.

Eclipse BPEL Designer is still under development. For example, direct XML code manipulation and debugging support are yet to be included. The Eclipse BPEL project web site[3] also includes the project milestone plan.

At the moment, only limited online tutorials appear to be available for the Eclipse BPEL Designer. For example, there are some videos which show how to perform certain tasks, such as creating a new BPEL project or partner link.

Whenever a new partner link type is created in Eclipse, it is automatically added to the WSDL file of the project. The file can import other WSDL files specifying the port types to which the partner links refer.

In addition, Eclipse includes a tool for testing Web Services, the Web Services Explorer. The tool is able to connect to a Universal Description Discovery & Integration (UDDI) [105] registry. Then, it is possible to, for example, publish web

[3]Eclipse BPEL Project. Available at `http://www.eclipse.org/bpel/`. Accessed on 2008-4-30.

services in the registry. It is also possible to invoke the operations of web services based on their WSDL files. However, in order to invoke an operation, a supporting endpoint reference must be available. Otherwise, it is only possible to browse the contents of WSDL files using the Service Explorer.

2.1.4 Other Tools

Several BPEL development tools can be found on the Internet in addition to those presented above:

- Intalio|BPMS CE [4]
- Oracle BPEL Process Manager [5]
- Cape Clear 7.5 ESB [6]
- Fiorano BPEL [7]
- BizZyme BPEL Engine [8]
- eClarus [9]

Some of the tools are available as limited complimentary editions, and some offer an evaluation period after which the product must be purchased to continue use. In most of the cases, registration is required to obtain a trial version.

2.2 Semantic Web Service Technologies

This section reviews the previous advances in semantic web service research. It starts with a brief overview of the most popular languages for semantic web service descriptions.

2.2.1 Semantic Service Description Languages

Web service descriptions are typically formulated in WSDL (Web Services Description Language) [8], which only describes the syntax of communicating with a web service. Semantic description languages are required for expressing the meaning of service interface elements, such as operations and data types.

[4] Available at http://bpms.intalio.com/. Accessed on 2008-4-30.
[5] Available at http://www.oracle.com/technology/products/ias/bpel/index.html. Accessed on 2008-4-30.
[6] Available at http://www.capeclear.com/. Accessed on 2008-4-30.
[7] Available at http://www.jvl-fr.com/spip/IMG/pdf/fiorano_bpel_brochure.pdf. Accessed on 2008-4-30.
[8] Available at http://www.creativescience.com/. Accessed on 2008-4-30.
[9] Available at http://www.eclarus.com/. Accessed on 2008-4-30.

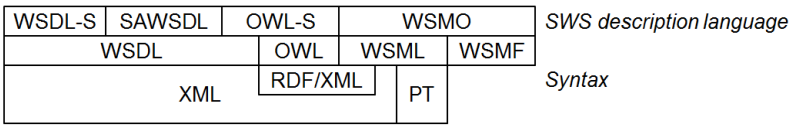


Figure 2.1: OWL-S includes only an XML-based syntax, while WSML, which is used by WSMO, comprises XML- and RDF-based syntaxes as well as a human-readable plain-text (PT) syntax.

The combinations of semantic descriptions and web services are called semantic web services (SWS) [62]. Several approaches have been proposed for adding semantic descriptions to web services. Figure 2.1 summarizes some of the most notable approaches and their underlying methodologies. In particular, OWL-S ("OWL for Services") [53] and WSMO (Web Service Modeling Ontology) [87] have attained wide acceptance as semantic web service description languages. These alternative approaches will be presented in more detail in the sequel.

OWL-S

Web Ontology Language (OWL) [61] is designed for formulating semantic descriptions. It extends the RDF (Resource Description Framework) [43] syntax. While RDF features a plain-text syntax, N-Triples, it is mainly applicable as a shorthand, and RDF/XML (XML syntax for RDF) [86] is the recommended syntax for information exchange. However, as such, OWL is a general purpose language for ontologies and provides no standard constructs specifically designed for web service definitions.

OWL for Services (OWL-S) [71] is an OWL ontology for web services that has attained wide acceptance in the research community. OWL-S makes it possible to semantically describe service input and output parameters, service preconditions, and conditional effects.

OWL-S service descriptions consist mainly of three types of elements, namely *ServiceProfiles*, *ServiceModels*, and *ServiceGroundings*. This dissertation refers to instances of these three element types respectively simply as *profiles*, *processes* and *groundings*. Instances of other classes in the OWL-S ontology will be referred to by writing the class names in lowercase italic letters. Some of the most essential classes of the OWL-S ontology and the possible property relations between their instances can be observed in Figure 3.10. Both *profiles* and *processes* are able to describe the inputs, outputs, preconditions and effects of web services. While *profiles* are mainly intended for use in web service discovery, the *processes* provide the information necessary to invoke the services. However, an atomic OWL-S *process* can be executed only if it is grounded to an underlying web service through an OWL-S *grounding*.

While OWL-S *processes* may include preconditions that specify when the pro-

cesses can be executed, the effects of *processes* are described through *results*. Each *process* may have zero or more *results*, each with different conditions and effects. Thus, a *process* may have different effects depending on the situation in which the *process* is executed.

However, OWL-S defines no syntax for the actual condition and effect expressions [53]. Instead, OWL-S defines only place-holders for the expressions and allows the use of any expression language. The languages may incorporate both plain text and XML-based formats. Some potential expression languages are XML-based, such as SWRL [30], while others are textual, such as SPARQL [79]. In addition to selecting the expression language, the details of applying the language are an unrestricted design choice. For example, the SPARQL query language includes four different query types, which yield different types of results.

Groundings provide the links from semantic web service descriptions to the underlying syntactic web service interfaces. Thus, a *grounding* describes how to access the service and communicate with it. Although several types of *groundings* can be defined, WSDL groundings are the most widely used grounding type [53]. WSDL *groundings* map atomic *processes* to WSDL operations [53]. Hence, modifications to a syntactic web service interface typically necessitate modifications to the *groundings* included in the semantic service description as well. Nevertheless, groundings may also be provided dynamically [53].

Unfortunately, each *service* may contain only one *process*, and a *process* mainly corresponds to a single web service operation [54]. Because OWL-S currently contains no construct for grouping *processes* [54], and a typical web service interface contains several operations, a complete semantic description of a web service interface should obviously define several different *services*.

OWL-S has established a wide user community in the scientific field. For example, Paolucci et al. [73] have developed an OWL-S virtual machine capable of parsing OWL-S service descriptions and executing the processes. Vaculin and Sycara [109] have further developed the virtual machine and extended it with an event mechanism, so that it is possible to monitor the execution state of an OWL-S *process*. Paolucci et al. [73] use the term DAML-S, as which the previous versions of OWL-S were known. However, in this dissertation, the term OWL-S is used in generally referring to all versions of the specification.

OWL-S API [72] makes it possible to execute OWL-S *processes* from Java code. A considerably newer version of the OWL-S API, which primarily supports OWL-S version 1.2, is available at [108]. In particular, the API provides facilities for loading OWL-S descriptions and creating Java object models representing their contents. The object models include methods for retrieving OWL-S *processes* by their URIs, setting values to their *inputs*, executing the *processes* and retrieving the *output*

values.

WSMO

Web Service Modeling Ontology (WSMO) is a widely-accepted alternative to OWL-S for formulating semantic web service descriptions. While OWL-S is based on OWL and requires the use of some other language in formulating web service conditions and effects, WSMO descriptions are formulated entirely in a specific formal language, WSML (Web Service Modeling Language) [87].

WSMO is conceptually based on the Web Services Modeling Framework (WSMF), which aims to describe the interfaces of web services instead of their internal logic [21]. Instead of OWL, WSMO relies on the Web Service Modeling Language (WSML) [120]. Actually, WSML is a family of languages featuring different syntaxes. Because the human-readable syntax of WSML is problematic for automated processing, XML and RDF based syntaxes, WSML/XML and WSML/RDF, have also been developed [120]. WSML/RDF allows the representation of WSML descriptions as RDF graphs. However, when WSML specifications are exchanged between computers, the WSML/XML syntax is the most viable option [120]. Roman et al. [87] claim that WSMO has several advantages over OWL-S. For example, WSMO allows several interfaces to be defined for a web service, while OWL-S allows only one [87].

Wang et al. [116] perceive that WSMO lacks a sufficiently formal specification and believe that this might cause developers to understand its semantics differently. Thus, they have developed a formal description of the syntax and semantics of WSMO.

A WSMO API for Java entitled wsmo4j has been developed as an open source project[10]. The API also supports the SAWSDL specification. For example, it enables programmatic addition of semantic annotations to WSDL files. However, it appears that the API only supports WSDL 2.0. The project homepage provides code examples of how ontologies and WSMO service descriptions can be programmatically created using the API.

SAWSDL

SAWSDL (Semantic Annotations for WSDL and XML Schema) [20] provides a standardized means of enriching web services with semantic descriptions. The Web Services Description Language, WSDL [8], is commonly used for describing web service interfaces. However, as such, WSDL is unable to describe the semantic features of web services, such as the goals of web service operations or the meanings of their inputs and outputs.

[10] Available at http://wsmo4j.sourceforge.net/index.html. Accessed on 2013-5-15.

SAWSDL defines an annotation mechanism for linking semantic descriptions to the most closely corresponding WSDL elements and also makes it possible to specify the translations between the XML message structures and the semantic concepts describing them. In particular, SAWSDL defines three extension attributes to the basic WSDL syntax: *modelReference*, *liftingSchemaMapping* and *loweringSchemaMapping*. In the sequel, the three attribute types are called model references, lifting schema mappings, and lowering schema mappings instead of using the camel-back notation.

Model references define the semantic concepts, such as OWL classes, corresponding to the referencing WSDL or XML Schema entities. In specifying operation input and output message structures, WSDL documents typically refer to XML Schema types and elements. Lifting schema mappings define the translation from the referencing XML structures to the semantic concepts expressed in the semantic language, while lowering schema mappings define the opposite translation to the XML structures. The transformation language used in schema mappings is unrestricted, although XSLT [9], which is part of the Extensible Stylesheet Language Family (XSL)[11], is frequently used.

All of the three extension attributes accept both single URIs and sets of URIs as their values. For example, a model reference may point to several semantic concepts, in which case all of them are considered to describe the element. In addition, SAWSDL provides an extension element with the name *attrExtensions* to maintain compatibility with WSDL 1.1. The extension element can only be used in annotating WSDL 1.1 operations, which accept no extension attributes. The *modelReference* attributes corresponding to the operations can then be added to the *attrExtensions* elements.

While SAWSDL annotations typically reference concepts in OWL ontology models, SAWSDL mandates the use of no specific semantic modeling language. In SAWSDL, references to semantic descriptions are made by using URIs, and the URIs may point to semantic concepts defined in any semantic modeling language.

SAWSDL has been developed based on the preceding WSDL-S [2] language. On the one hand, the two languages share several similar features. For example, WSDL-S and SAWSDL both define a *modelReference* extension attribute. On the other hand, SAWSDL also differs considerably from WSDL-S. For example, only WSDL-S defines extension elements for adding precondition and effect annotations to web service operations. Nonetheless, operation preconditions and effects can also be expressed through the *modelReference* attribute. Hence, it seems that omitting these constructs only simplifies the SAWSDL specification as well as removes potential redundancies and inconsistencies in the created web service definitions.

[11] `http://www.w3.org/Style/XSL/`

SAWSDL is noncommittal regarding the semantic description language used or the type of concepts referenced through SAWSDL annotations. Moreover, SAWSDL itself defines no rules to how the semantic annotations are to be used or interpreted [54]. Hence, Martin et al. [54] provide recommendations on using SAWSDL with OWL-S. For example, since WSDL operations most directly correspond to OWL-S processes, the SAWSDL annotations attached to WSDL operations should refer to OWL-S processes [54]. Iqbal et al. [34] apply SAWSDL annotations in linking WSDL operations to SPARQL [79] expressions defining the operation preconditions and effects.

While semantic annotations in SAWSDL refer to semantic concepts through URIs, the actual method of retrieving the semantic descriptions is unspecified. For example, the URIs may resolve to external documents. Alternatively, XML-based semantic descriptions, such as OWL-S, may be embedded into the WSDL document [20]. Indeed, as WSDL descriptions accept extension elements, the models may also be directly embedded in the SAWSDL description, provided that they are expressed in XML [20]. For example, OWL is based on XML, and the WSML family of languages, which is used by WSMO, also includes an XML variant.

Although SAWSDL is a rather new specification, it is already supported by software tools, such as an SAWSDL library for Java, SAWSDL4J[12]. Unfortunately, the current version of the library appears to be based on a preceding version of the SAWSDL specification, and it appears to be incompatible with the latest version [20]. The main reason for the incompatibility is that the two versions of the specification use different namespace URIs for identifying the SAWSDL namespace. On the other hand, the source code of the library is also publicly available, and thus it is easier to overcome such difficulties in using it. Woden4SAWSDL[13] is another open source Java library which provides an object model for semantically annotated WSDL documents. However, it only supports WSDL 2.0, whereas SAWSDL4J also supports WSDL 1.1.

2.2.2 Semantic Web Service Orchestration

UDDI registries [105] permit web service providers to publish their services. However, Sycara et al. [99] point out that the UDDI standard is insufficient for discovering web services based on their capabilities. Thus, they have developed a DAML-S/UDDI Matchmaker, which allows services to be discovered based on semantic descriptions written in OWL-S. The system contains a communication module which receives semantic service advertisements. Once an advertisement is received, it is

[12]Available at http://sawsdl4j.sourceforge.net/. Accessed on 2013-5-15

[13]Available at http://lsdis.cs.uga.edu/projects/meteor-s/opensource/woden4sawsdl/index.html. Accessed on 2009-11-11.

sent to a translator module which converts it to a UDDI service record and stores the record in a UDDI registry. Finally, the advertisement is stored in a matching engine together with the UDDI service record identifier. When the system receives a semantic web service request, the matching engine compares it to the stored semantic service advertisements. The comparison algorithms find the service descriptions that semantically match the request. Furthermore, the found services are graded based on how accurately they match the request. Because the system maintains associations between the stored advertisements and the corresponding UDDI service identifiers, the services associated to the matching descriptions can be retrieved from the UDDI registry based on their identifiers [99].

Çelik and Elçi [6] propose an alternative approach to using OWL-S in web service discovery. In their approach, a semantic search agent assists a user in finding an appropriate web service from a UDDI registry. The user must first supply the agent with search terms, which the agent uses to find potentially matching web services in the service ontology. According to Çelik and Elçi [6], the agent also compares the inputs and outputs of the services to those specified by the client. Çelik and Elçi [6] claim that the matching algorithm implemented by Sycara et al. [99] operates in UDDI, while in their approach the search algorithm is implemented completely in the semantic search agent, which allows the UDDI registry to function traditionally. Nevertheless, it appears that the system developed by Sycara et al. [99] is implemented as a separate component that uses a UDDI registry but has no impact on its internal operation. Thus, the UDDI registry could be operated independently of the matching system, and it can be argued to function traditionally also in the Matchmaker developed by Sycara et al. [99]. Hence, the two systems appear to be based on somewhat similar principles. On the other hand, the approach proposed by Çelic and Elçi [6] contains a search term enhancement phase which precedes the actual semantic matching phase and is apparently omitted in the former approach. In the search term enhancement phase, the search terms entered by the user are enriched by finding synonyms and applying semantic is-a relationships.

Çelik and Elçi [7] have applied the semantic search agent approach in implementing a semantic web service search system. The user enters required input and output types to the system, and the system displays the results as a list of web services. The system scores the services according to the semantic similarity of the inputs and outputs to the search terms.

Song et al. [96] describe how SAWSDL can be used in automatically determining services that semantically match search criteria specified by a user. They find the SAWSDL annotation method simple and flexible. However, Iqbal et al. [34] claim that SAWSDL alone is inadequate for specifying web service preconditions and post conditions, which are essential in determining whether a web service matches the user

goals. Thus, Iqbal et al. [34] propose an approach that uses SPARQL [79] queries to specify web service pre- and post conditions in SAWSDL files. In their method, the SAWSDL *modelReference* attribute is used to refer to the SPARQL queries. For example, the outcome of a service is represented by a SPARQL CONSTRUCT query and the user goal is formulated as a SPARQL ASK query. The ASK query is evaluated over the RDF graph that results from evaluating the construct query. If the result is true, the service fulfills the goal.

Systems encompassing several heterogeneous web services may involve interoperability problems, which can be solved by using semantic web service descriptions [39]. For example, Delamer and Martinez Lastra [13] present an SOA approach in which service descriptions are augmented with semantic information to allow automated service selection and invocation in the domain of manufacturing systems. Nevertheless, somewhat similar approaches are applicable in the medical domain as well [44; 31]. In addition, Sasa et al. [88] present an approach applying agent technology and ontologies to automate decision-making in business system tasks that would otherwise require human participation. This dissertation will focus on the application of semantic web services in production system automation.

Various authors have proposed methods of combining the use of BPEL and semantic service descriptions, and these methods have some features in common with those proposed in this dissertation. For example, Verma [111] proposes defining semantic templates in BPEL processes. The semantic web service descriptions are achieved through the use of SAWSDL and OWL. However, while this dissertation applies OWL-S in describing the functionality of web service operations, Verma [111] defines functional concepts for this purpose. The functional concepts are then referenced in the SAWSDL annotations added to WSDL operations similarly as Martin et al. [54] suggest linking WSDL operations to OWL-S Processes. Indeed, the functional concepts defined by Verma [111] appear to encompass OWL-S Processes: for example, both include inputs, outputs, preconditions and effects.

Another approach that considerably resembles the methodology proposed in this dissertation is the BPEL for Semantic Web Services (BPEL4SWS) [65] language, which extends the standard WS-BPEL 2.0 specification. Unlike plain BPEL processes, BPEL4SWS processes are independent of the WSDL descriptions of partner web services. Instead, partner service requirements can be expressed semantically. Hence, BPEL4SWS resembles the approach proposed by Verma [111]. Nevertheless, the service orchestration approach proposed in this dissertation is different from both of the aforementioned methodologies in that it only uses the standard WS-BPEL 2.0 language without any extensions. Instead, in the proposed approach, BPEL processes refer to WSDL interfaces, and, because the interfaces are annotated with semantic information, the proposed framework can automatically modify the BPEL

processes to use different WSDL interfaces depending on the results of semantic service selection.

In addition to semantic descriptions, other methods of modeling the states of web services have been developed. In particular, the Web Services Resource Framework (WSRF) [121] facilitates accessing stateful resources through web services. Instead of semantic models, WSRF uses plain XML schemas to describe resource properties. Clients can query and manipulate resource states through standard operations specified in WSRF as well as through service-specific operations. WSRF has been applied, for example, in the domain of grid services [117]. Unlike WSRF, semantic web service descriptions set no requirements on the web service WSDL interfaces. In particular, web services are not required to provide the special operations defined in WSRF for querying and updating resource states. Furthermore, syntactic differences in web service interfaces and the representation of resource states are considerably less relevant when semantic descriptions are applied. While the approach requires that semantic descriptions on available web services and domain resources are created and made available, no changes to web service implementations are required.

In addition to BPEL, other techniques have also been developed for modeling process workflows. For example, Petri Nets-based approaches have been applied in cross-organizational workflow modeling [38]. Work on web service orchestration using Petri Nets and Timed Net Condition/Event Systems, a Petri Nets-derived formalism, can be found respectively in [63] and [77].

2.2.3 Automatic Web Service Composition

While OWL-S is suitable for formulating process prescriptions, BPEL can be considered more standardized [55]. For example, Martinek et al. [55] apply BPEL processes in the domain of enterprise systems and dynamically generate mediator services that automatically resolve syntactical incompatibilities between service interfaces. The approach eliminates the need to manually prescribe data transformation scripts in, for example, OWL-S *groundings*. However, the task of automatically prescribing the appropriate data transformations appears rather complex. Moreover, the approach is noncommittal regarding the automatic composition of the overall process prescription, which is the main focus of this dissertation.

Semantic web service description languages facilitate automated web service composition. However, the majority of publications focus mainly on determining service suitability according to the input and output types, which may be due to the neutrality of OWL-S with regard to expressing service conditions and effects [89]. While concentrating on input and output types may be sufficient with information-providing web services, conditions and effects are essential in describing world-altering web services. For example, in the factory automation domain, web services

typically represent production devices, such as conveyors. Therefore, the focus is on world-altering web services.

Regardless of the semantic language selected, composing semantic web services to achieve a goal essentially entails solving a planning problem. A planning problem can be formulated as the tuple $\langle S, s_0, g, A, \Gamma \rangle$, where S is the set of all possible states the system may have, s_0 is the initial state, g is the goal state, A is the set of actions, and $\Gamma \subseteq S \times A \times S$ is the state transition relation. Γ describes which actions may be applied to each state, and to which state the system will transition as a result of applying an action [85].

For example, Hatzi et al. [27] present a framework that converts web service composition tasks into planning problems expressed in PDDL (Planning Domain Definition Language) [60]. Once a solution plan has been found, the framework converts it into an OWL-S composite process description. The framework translates atomic OWL-S processes to planning operators, from which it derives the set of actions.

Translating OWL-S descriptions to PDDL is advantageous because PDDL is widely used in AI planning and supported by several planners. In addition, due to the influence of PDDL on the development of OWL-S, translations between the two languages can be automated [123].

To reduce the complexity of planning, hierarchical task network (HTN) planning requires that the planning domain description includes a set of method definitions that specify how complex tasks can be hierarchically decomposed into smaller tasks [94]. The planning problem can then be specified as a list of tasks to perform given a certain planning domain and initial state. The planner attempts to solve the problem by applying the methods to each task in the task list. Thus the planner hierarchically breaks down each task into a partially ordered set of subtasks, until a sequence of atomic planning operators is obtained, which corresponds to a solution plan [94]. For example, the SHOP2 planner [94] applies HTN planning.

To find a sequence of web service operations that leads to the fulfillment of a goal, a software agent may build a graph rooted at the initial domain state, so that each node represents a domain state resulting from different operation invocations [89]. If each operation is considered to result in a different domain state, the generated graphs are trees [89]. Unfortunately, the number of web services and operations available tends to dramatically impair web service composition efficiency [125].

Sbodio et al. [89] demonstrate the use of OWL-S and SPARQL in selecting web services capable of achieving a goal. In particular, SPARQL ASK type queries, which evaluate to Boolean values, are applicable for expressing goals, while CONSTRUCT type queries allow the expression of web service operation conditions and effects [89]. While the SPARQL/Update language actually appears more suitable for expressing

effects, it is still at a rapidly evolving development stage, and hence Sbodio et al. [89] represent effects using CONSTRUCT queries.

This dissertation will focus on the use of OWL-S with SPARQL and SPARQL/Update as the expression languages. The approach applies the guidelines presented in [89] for representing the world states with RDF graphs. Nevertheless, while [89] focuses on service discovery, or selection, and only outlines the possibility of sequential service composition, this dissertation elaborates on the latter. Furthermore, this dissertation presents an approach to updating the semantic model of the world state based on OWL-S, SPARQL, and event notifications.

The use of SPARQL in this dissertation differs from that presented in [89]. Mainly, instead of combining conditions and effects in SPARQL CONSTRUCT queries, SPARQL ASK queries are used in expressing conditions and SPARQL/Update statements in expressing effects. SPARQL/UPDATE appears preferable for expressing effects because SPARQL CONSTRUCT queries can only express 'positive effects', i.e. statements that are true after executing a process. In contrast, the INSERT/DELETE statement of SPARQL/Update also makes it possible to specify 'negative effects', i.e. statements that are false after executing a process, in a single expression. While Sbodio et al. [89] point out that one could use two CONSTRUCT queries to express both positive and negative effects, OWL-S appears to provide no constructs for discriminating between expressions that represent positive and negative effects.

While domain ontologies are essential in modeling production systems [56], there is no definitive standard for structuring such ontologies. For example, Ferrarini et al. [22] present the *PABADIS'PROMISE* approach, which includes a meta-model for domain ontologies. Furthermore, Ferrarini et al. [22] demonstrate using the model with agent-based control systems. Somewhat similarly, Uddin et al. [107] use a domain model in optimizing a web service -based production system. Although the domain ontology used in [107] is not based on a meta-model, the ontology is divided into layers, so that specific use cases extend the concepts defined in the template level. Martinez Lastra and Delamer [56] survey the current approaches of using ontologies in describing manufacturing systems.

To reduce the effort involved in developing and maintaining domain ontologies, Segev and Sheng [90] propose an approach to automatically creating ontology models for web service -based systems. However, it seems debatable whether an automatically-generated model could be used in developing an expert system, such as the optimization system presented in [107].

The Open, Object-Oriented kNowledge Economy in Intelligent inDustrial Automation (OOONEIDA) project [114] facilitates the reconfigurability of production systems through the concepts introduced in the IEC 61499 standard [33]. In par-

ticular, OOONEIDA perceives the concept of an IEC 61499 function block as a vehicle for intellectual property inserted into a reconfigurable device or machine. Furthermore, Vyatkin et al. [114] propose building a knowledge repository using a semantic language such as OWL. As the older IEC 61131-3 standard [32] still has much wider support, Dai and Vyatkin [11] propose an automatic transformation from IEC 61131-3 PLC programs to IEC 61499 function blocks. The transformation includes first importing the PLC programs to an OWL knowledge base and then applying rules formulated as SQWRL [69] queries to map the OWL instances to another knowledge base containing function block descriptions [11].

This dissertation applies the service-based approach presented in [57], in which a specialized web service carries out semantic web service composition and planning. To achieve greater decentralization, the domain ontology, which includes product definitions, is also hosted by another specialized web service. The two web services will be called respectively the Service Monitor and the Ontology Service.

A more distributed service composition framework has been proposed in [104]. However, the approach presented in [104] relies on more sophisticated web services, service agents, that combine web services and agent technology.

2.2.4 Semantic Domain Modeling

Controlling a manufacturing system requires developing a comprehensive knowledge representation of the system. An ontology-based knowledge representation of a factory automation system makes it possible to reuse the domain model in several autonomous software agents [58]. The domain model must describe the different processes performed in the production system and the services that offer the processes [58]. OWL (The Web Ontology Language) [61] is currently the de facto standard for representing ontologies in a machine-interpretable format. For example, the OWL-S [53] ontology for web services is based on OWL.

In a flexible manufacturing system, whose devices expose web service interfaces, the domain model must be populated at system start-up and continuously updated at run-time. To this end, a client application may poll the state of the web services by using the request-response type operations in the service interfaces [106]. However, the periodic polling introduces a steady overhead even during periods when no changes occur in the system. Moreover, the polling period must be set sufficiently short to reduce the risk of using outdated information.

The approaches presented in Section 3 are based on the assumption that there exists an OWL model describing the controlled manufacturing system. The OWL model contains statements describing the most current state of the system and thus allows optimization decisions to be made. Such a knowledge-based approach to flexible manufacturing system control reduces the overhead of raw data processing

and enables different software agents to interact using a common knowledge base [106]. In the approaches presented in this dissertation, the OWL model contains the concepts to which the service WSDL files refer through SAWSDL annotations. In the sequel, the OWL model describing the system state will be called the domain model.

Moser and Biffl [64] investigate semantic integration between different ontologies and present an approach for maintaining a common ontology model to which tool-specific ontologies are mapped. The approach makes it possible for different stakeholders to use their own local data models while still allowing the validation of the entire combined runtime model [64].

Another approach to integrate different domain ontologies is to use a common adaptable base ontology, which the domain specific ontologies extend [107]. Such a hierarchical domain ontology can be constructed using a top-down or a bottom-up approach or a combination of the two [101]. While using a common base ontology facilitates the integration between different knowledge models [101], it considerably restricts the development of domain-specific ontology models.

Loskyll et al. [51] point out that when web services are mapped to concrete devices in the plant model, the service preconditions and effects make it possible to automatically update the plant model. Similarly, one of the domain model update approaches presented in this dissertation is based on the semantic web service preconditions and effects.

The use of domain ontologies extends to several other application domains than factory automation. For example, Evchina et al. [19] apply domain ontologies in the control of automated systems installed into modern apartments. Tan et al. [100] propose a general context model for businesses that appears independent of any particular domain. Similarly to [107], in [100] the classes and properties in the main ontology are shared by the more specific context models. However, Tan et al. [100] point out that OWL alone is insufficient to express complex business rules involving, for example, mathematical computations, and that such rules can be expressed in SWRL [30].

In addition to the domain status description, the modeling of the available web services is an equally essential requirement for reasoning on how the domain status can be changed. In particular, the modeling of service conditions and effects is vital for such decision-making. Furthermore, Delamer and Martinez Lastra [12] point out that the effect descriptions enable the domain model to be updated as services are invoked.

De Virgilio [113] proposes a meta-model for web services to facilitate the storage and querying of web service descriptions. As the approach presented in [113] is based on relational databases, thus allowing high data storage and access efficiency,

it focuses on a lower data abstraction level than this dissertation, which focuses on operating with OWL models.

This dissertation discusses the updating of a common production system OWL model, so that the model remains synchronized with all relevant changes in the physical system. The classes, properties, and the number of OWL individuals in the model remain constant, while the assertions that describe the current world state are updated. Although the approach has been primarily developed to be applied in production systems, it can be utilized in any scenario involving web services that send event updates.

In this dissertation, all OWL models are expressed in OWL DL, which corresponds to description logics [61]. In description logics [16], a knowledge base consists of a set of intensional assertions called the TBox and a set of extensional assertions called the ABox [5]. The TBox contains the general description on concepts and relations, while the ABox contains the assertions on individual objects [5]. Thus, updating a production system OWL model essentially involves updating the ABox. Nevertheless, the Abox and TBox may actually reside in the same OWL model.

Legat et al. [46] demonstrate the use of a semantic production system model without the assumption that the devices are modeled as web services. They propose automatically composing PLC code based on a semantic domain model, which principally corresponds to the automatic composition of web services discussed in this dissertation.

2.2.5 OWL-S Generation

While detailed guidelines for adopting semantic web services in the factory automation setting exist [57], the formulation of semantic descriptions currently forms a considerable threshold to the industrial application of semantic technologies. For example, Puttonen et al. [84] use the manufacturing system described in [112] as an application scenario for semantic web service composition. The system includes 29 instances of a conveyor service and five instances of a workstation service. To apply semantic web service technologies to the system, it is necessary to create an ontology model of the system and a semantic description of each of the 34 web service instances. The creation of the semantic service descriptions includes modifying both the service WSDL file and the OWL-S document. While the process can be expedited through meticulously planned copy-paste and search-and-replace techniques, it typically results in a considerable delay as well as errors that occur at system run-time and are difficult to trace. Hence, mitigating the effort in producing the complete semantic web service descriptions should considerably reduce the threshold for the industrial adoption of semantic web services in factory automation.

Although the approach presented in this dissertation somewhat increases the

modifications required to the service WSDL files, it completely automates the creation of service OWL-S description documents. The proposed approach is primarily intended to be used by an autonomous software agent responsible for service composition based on a goal that is expressed in terms of an OWL domain model. Therefore, the approach especially focuses on service conditions and effects to the world state, whereas most current literature appears to focus on composing services based on their input and output type compatibility possibly augmented with some service classification criteria.

The service composition framework presented in [76] automates the generation of OWL-S descriptions for web services. Furthermore, it automates even the generation of XSLT transformations between the XML data elements exchanged by the services and the corresponding semantic concepts. In addition, the OWL-S API [108] includes a sample GUI application for generating OWL-S descriptions for WSDL operations. However, the sample application requires the user to manually specify any XSLT transformations and omits service preconditions and effects.

OWL-S incorporates a rather complex syntax, which can form a considerable learning barrier. Therefore, Timm and Gannod [102] have developed a methodology that facilitates the adoption of OWL-S by exploiting the more familiar UML language. Their approach involves first creating a UML class diagram representing the service ontology using a UML editing tool. For this purpose, they have developed a custom UML profile, in which classes are tagged with stereotypes. The stereotypes indicate to which OWL-S constructs the classes are mapped. The diagram is saved in XMI [122] format and processed by a conversion application. The conversion from XMI to OWL-S is performed by applying XSLT transformations. Figure 2.2 illustrates the approach. The conversion result is an OWL-S description of the service without a *grounding*. Timm and Gannod [103] have further developed the approach by creating a software tool which also generates the OWL-S *grounding*. However, the user is required to specify, for example, a mapping between WSDL operations and OWL-S processes through a graphical user interface [103]. In addition, the tool developed by Timm and Gannod [103] applies the OWL-S API [72] in executing OWL-S *processes* and allowing the user to monitor the progress.

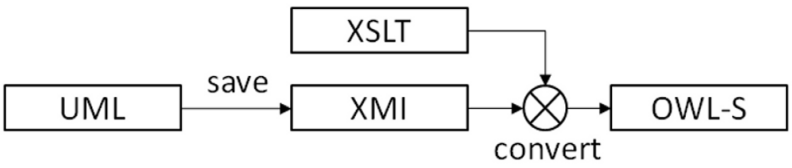


Figure 2.2: UML diagrams can be automatically converted to OWL-S descriptions by applying XSLT transformations to the saved XMI files.

Grønmo et al. [26] point out that OWL-S descriptions are typically rather verbose and difficult to read, which may impede an engineer from comprehending them. Hence, they propose using transformations in both directions between OWL-S and UML to facilitate web service composition. The approach involves first designing the composite process in UML. Then, the individual tasks in the process are transformed into semantic descriptions, which are used in semantic discovery of matching services. The semantic descriptions of the matching services are transformed into UML for manual examination and selection. Finally, the composite process is constructed from the selected services as a UML model, and a semantic web service description is generated from the model. Although the approach basically sets no restrictions on the used semantic language, and Grønmo et al. [26] describe how WSMO [87] could also be applied, they have only implemented transformations between UML and OWL-S. Similarly to Timm and Gannod [102], they have developed their own UML profile to be used in the transformations.

Yang and Chung [124] present a UML to OWL-S conversion method that additionally extracts service composition information from UML state diagrams.

Kim and Lee [41] also propose a method for converting UML diagrams to OWL-S *processes*. Their method is similar to the one proposed by Timm and Gannod [102] in that UML models are first saved in XMI format and then converted to OWL-S by applying XSLT transformations. Nevertheless, their method, in addition to using class diagrams for representing domain ontologies, also uses sequence and activity diagrams for representing business process behavior. Kim and Lee [42] compare their method with other proposed approaches to converting UML diagrams to OWL-S models, such as that of Grønmo et al. [26]. Kim and Lee [42] point out that their method also allows modeling of complex processes which comprise several control constructs, whereas most of the earlier methods mainly focus on atomic processes. In addition, experiments carried out by Kim and Lee [42] show that an automatic conversion from UML to OWL-S following their approach seldom results in a higher number of elements than a manual conversion performed by a human expert.

While it is straightforward to create a transformation script that converts XML structure into RDF [20], it is considerably more difficult to define the translation in the opposite direction due to the numerous different ways of serializing a single RDF model to XML [54]. Hence, the XSLT transformation scripts used in lowering schema mappings should consider all possible alternatives of serializing the RDF data. As a solution, the SAWSDL specification [20] suggests using hybrid transformations consisting of two scripts: the first one is a semantic query language expression, such as a SPARQL query, and the last one is an XSLT transformation. As query languages such as SPARQL define only one way of serializing the results into XML, the XSLT transformation is straightforward to write. Unfortunately, the OWL-S

WSDL grounding allows only a single XSLT transformation script to be defined for each variable through the datatype property *xsltTransformationString*, and there is no corresponding property for specifying a preprocessing expression, such as a SPARQL query. Nevertheless, a simple XSLT transformation may be sufficient even for lowering schema mappings, since both Timm and Gannod [103] and Pi et al. [76] present software tools able to automatically generate lowering XSLT transformation scripts based on a mapping from semantic concepts to XML schema entities.

Paolucci et al. [74] examine the similarities between OWL-S and SAWSDL and conclude that SAWSDL annotations provide sufficient information for generating OWL-S service descriptions. However, SAWSDL provides no information allowing the derivation of preconditions and effects for the OWL-S *processes* [74].

To allow lowering XSLT scripts to prepare for only a very restricted set of formatting options, the software tool that processes the OWL-S definitions to invoke web services can preprocess the input data to the XSLT scripts. For example, the software tool presented in this dissertation preprocesses the input data so that an XSLT script can assume the input RDF graph to be only a small subgraph rooted at the OWL individual representing the input value. Hence, there is only one possible XML serialization for the input RDF data. The preprocessing approach has been inspired by the OWL-S WSDL grounding implementation included in the OWL-S API [108].

The approach proposed in this dissertation also includes generating the required XSLT scripts unless they have been explicitly specified. Nevertheless, unlike the tools presented in [76] and [103], the software tool presented in this dissertation requires no user interaction during the OWL-S generation process. Instead, the tool is implemented as a web service. In addition, the approach presented also considers the service conditions and effects as well as the data interfaces.

The purpose of SAWSDL annotations in the approach presented in this dissertation is somewhat different to their conventional use in the literature. For example, Sellami and Boucelma [91], as well as Hobold and Siqueira [28], apply SAWSDL annotations in service discovery and composition. Nonetheless, the generality of the annotations allows them to be applied in a wider range of use cases, and Opdahl [70] applies SAWSDL annotations in improving the interoperability of different software modeling languages. This dissertation proposes using SAWSDL annotations as guidelines for generating the service OWL-S descriptions for web services representing manufacturing devices. In particular, the approach uses SAWSDL annotations to generate the OWL-S *process* conditional effects and the XSTL transformations used in the OWL-S *groundings*.

2.3 Cloud Computing

There are different types of cloud computing. In Infrastructure-as-a-Service (IaaS) the leased resources are virtual computer infrastructure [40]. More specifically, the resource units leased from IaaS clouds are virtual machines [97], which behave identically to actual servers connected to the internet. However, they are created through virtualization from actual servers. Other types of cloud computing include Software-as-a-Service (SaaS) and Platform-as-a-Service (PaaS). In SaaS, software vendors make their applications accessible over the Internet, while in PaaS the cloud systems provide platforms that allow software vendors to implement their applications. Then, the end users can access the applications over the Internet similarly to SaaS [45].

Public IaaS clouds are typically commercial enterprises from which virtual machines can be leased at certain prices [97]. The Amazon Elastic Compute Cloud, Amazon EC2[14], is a notable example of public IaaS clouds. Alternatively, organizations can create private clouds that are used internally and non-commercially [97]. The main purpose of private clouds is to share existing resources, rather than provide additional resources. On the other hand, private clouds may also use the resources of public clouds, and the combinations are called hybrid clouds [97].

Cloud computing toolkits, such as Eucalyptus [66] allow the creation of private and hybrid clouds. While there are no standard computing cloud interfaces, the private clouds created using the Eucalyptus software framework conform to the Amazon EC2 cloud interface and can be used with the same client tools [66].

For example, companies consisting of several departments can benefit from the effortless resource allocation enabled by private IaaS clouds. If each department were allocated physical servers, they would be idle for considerable periods of time. While reallocating physical servers to different departments might cause considerable amount of additional work, it is straightforward to dynamically start virtual machines and attach virtual storage volumes to the virtual machines with all the necessary software and data pre-installed.

Although many IaaS clouds support similar interfaces, starting an application in an IaaS computing cloud may be laborious due to the low-level nature of IaaS clouds [93]. Indeed, to lease a virtual machine, a client must first select the appropriate virtual machine image to use, and communication with a virtual machine instance is typically performed by logging in to the instance with a terminal program.

For example, the semantic web services orchestration framework proposed in [80] is implemented as a set of web services. The web services orchestrated using the framework can be hosted in resource-constrained embedded devices. In the orches-

[14]http://aws.amazon.com/ec2/

tration framework, the performance issues related to memory and CPU resources can be overcome by outsourcing some of the resource-demanding functions to the cloud. Considering service oriented architecture (SOA), it would be natural for the applications deployed in the cloud to provide web service interfaces. Fortunately, computing clouds can facilitate the dynamic deployment of web services, such as those forming the web service orchestration framework. Some web services may be needed only for a limited time, after which the computing resources reserved by them should be released. Moreover, deploying the services on physical server machines might require considerable effort in configuring and installing the hardware and software. The use of cloud computing is a more feasible approach, as it allows the dynamic creation of virtual machines for hosting the web services, thus reducing the number of actual computer systems required and the amount of idle resources.

While Cloud computing is a fairly recent paradigm, it is closely related to the more established concept of Grid Computing [23]. However, this research work will only focus on cloud computing.

2.4 Multi-Agent Systems and Robotic Agents

Systems composed of web services appear to have several common features with multi-agent systems (MAS). A MAS is modularly composed of several software agents [127] similarly as the web service orchestration framework presented in this research work is composed of individual web services. The agents co-operate to achieve complex goals. In MASs, individual agents may fail in performing their tasks, and ontologies can be used in diagnostics and error recovery [92].

The agents in MASs differ from web services in that agents integrally belong to a larger system in which they have a specified role, whereas web services more typically exist individually. Furthermore, the communication standards for web services are open, whereas agents typically rely on a specific communication infrastructure. In addition, the topologies of MAS presented in [127] are less relevant for web services, as web services can typically communicate with any other accessible endpoint over the internet. However, the main difference between agents and web services is perhaps in their nature. While agents actively pursue goals [127], web services are typically passive until receiving a service request.

The advantages of MAS include the ability to reach great decentralization and fault-tolerance. Furthermore, the roles of individual agents make it possible to formally validate such systems. Leitão and Rodrigues [47] demonstrate the use of Petri nets in formally validating production systems that are based on the MAS paradigm.

It is possible to develop systems combining both web services and agents. For example, Villaseñor et al. [112] have applied agents in implementing a decision-

support system for the orchestration of web service-based manufacturing systems.

The approach proposed in this research includes the assumption that the individual devices in a production system are abstracted as web services that perform little independent reasoning, while separate entities maintain a model of the world state and coordinate the devices based on the current state. Thus, the approach is somewhat opposite to certain approaches in robotics that aim to develop highly autonomous robots capable of performing independent decisions based on sensory data [98]. Nevertheless, the approach presented can also be used with devices equipped with advanced sensory and reasoning capabilities. For example, manufacturing system devices typically include sensors.

2.5 Summary of the State of the Art

The above discussion on related work shows that the application of semantic web service technologies is still far from standardized. Therefore, the next chapter will present a new methodology that builds on the work presented in this chapter.

While there exists a multitude of potential description languages and application approaches, none of them is straightforward to apply in practice. Moreover, in several cases, it appears questionable whether semantic technologies actually yield additional value over traditional syntactic technologies. For example, most of the techniques for selecting applicable semantic web services are based on examining class hierarchies, which basically constitutes mere categorization. In contrast, the next chapter will focus more on composing semantic web services based on their world-altering effects than on evaluating the semantic resemblance between the service parameters.

While there are several software tools capable of deriving OWL-S descriptions from WSDL documents, they appear to typically require direct user interaction. The next chapter will present an approach that eliminates the requirement by relying on SAWSDL annotations instead. Moreover, several of the current web service composition tools are user interface applications, whereas this research work focuses on a framework of co-operating web services, and the proposed service-composition tool is itself a web service.

The full potential of cloud computing is yet to be achieved as well. After resources have been dynamically leased from a cloud, their configuration and utilization still requires a considerable amount of effort. Moreover, the mere dynamic leasing of computing resources based on the current demand constitutes a considerable additional workload, unless it can somehow be automated.

3. METHODS

This chapter presents new methods that build on the related technologies presented in the previous chapter. First, Section 3.1 presents an approach to using BPEL in specifying workflow prescriptions for production systems. Then, Section 3.2 presents an approach of using a set of web services to orchestrate semantic web services through BPEL. Section 3.3 presents a semantic web service composition approach which eliminates the need of manually prescribing workflows in BPEL. Section 3.4 elaborates on alternative approaches to maintaining a domain model synchronized with changes in the production system state. As semantic web service descriptions are typically somewhat laborious to create, Section 3.5 presents a method of partially automating the process. Section 3.6 describes the details of a more advanced service composition approach. Finally, Section 3.7 presents a method of optimizing the use of resources leased from computing clouds.

The methods proposed in this chapter result in a framework of web service orchestration tools. The tools include a set of web services, of which Table 3.10 provides a summary.

3.1 Web Service Orchestration Using BPEL

Traditionally, industrial systems have been controlled by Programmable Logic Controllers (PLCs). With the advent of web service technologies, several research initiatives started investigating possible benefits that service oriented architectures could bring for the systems in production industry. In such systems, the production lines can be decomposed into smaller equipment pieces, so that each piece of the equipment is abstracted through a web service. The automatic composition of such services in order to fulfill customer needs allows the creation of loosely-coupled production systems, which are reconfigurable to accommodate new and possibly unforeseen needs [14].

Typically, a manufacturing line consists of a number of devices, each of which can be abstracted as a service provider. The devices interact to jointly constitute a more complex service. A manufacturing line may include several devices providing a similar service, such as a conveyor segment. However, these services might not be interchangeable. For example, different instances of a conveyor segment might have different physical locations, and thus, for a pallet, it is imperative that the

correct segment provides the service at each time. Moreover, the services typically have an internal state. For example, a conveyor segment may contain a pallet, in which case it is unable to load a new pallet until the old pallet is first unloaded. Thus, orchestrating a manufacturing line requires the ability to differentiate between different devices.

The Web Services Business process execution language (WS-BPEL) [3], which is abbreviated to BPEL in the rest of this dissertation, provides a standard manner of representing web service workflows in XML. It follows the prior BPEL for web services (BPEL4WS) language. BPEL makes it possible to orchestrate complex web services from simpler ones. Thus, with BPEL, it is possible to create a web service which performs some complex task through invoking the operations of some existing web services. BPEL models the workflow and the flow of data between the member services. These simple web services may remain unaware of each other. The interaction between separate web services occurs through partner links, which use the WSDL (Web Service Description Language) [8] files describing the communicating web services. As each BPEL process is a web service, it must have a WSDL file which describes the service. A BPEL process operates on a higher level than the WSDL files to which it refers. The WSDL files describe which operations can be invoked, and the BPEL process describes a sequence which uses the operations. [75]

In a BPEL process, the operations of individual web services do not have to be invoked sequentially. BPEL includes several control constructs which allow, for example, concurrent execution and the branching of execution.

To orchestrate a composite web service, in which each individual service plays a specific role, it would be necessary to create a set of instances of each service statically. As BPEL processes communicate with web services through partner links, the partner links must be initialized to web service instances before use. Then, it is possible to refer to those partner links when invoking operations. It is, for example possible to use the BPEL Assign activity to copy the endpoint reference information of a running service instance to a partner link. However, this requires that the actual service connection information is available at the time of writing the code of the BPEL process. On the other hand, different BPEL execution engines provide various methods of configuring partner link initialization at process deployment phase. For example, in the ActiveBPEL server environment discussed in Section 2.1.2, one can specify the endpoint references of the partner links in the process deployment descriptor (PDD) file. It is even unnecessary to know the actual URIs of the endpoint references at the time of creating the PDD file: according to the ActiveBPEL Designer help system, one can specify logical URIs in the PDD file and use the ActiveBPEL Server Administration Console to map the logical addresses to actual URLs subsequent to deploying the process.

Section 5.2 will exemplify BPEL-based web service orchestration.

3.2 Ontology-based Service Invocation

This dissertation experiments in representing the internal states of web services with semantic models. In the sequel, the web services that represent production equipment are called domain web services to differentiate them from the web services forming the proposed orchestration framework. In each web service orchestration scenario, the internal states of the domain web services are linked to the state of the production system, which is represented by a single semantic model.

Since the domain web services represent production equipment, the semantic model describing their internal states contains concepts representing objects in the factory floor as well as their features and relationships. Thus, the model may be called the equipment ontology or equipment model. The equipment ontology contains both a static part, which mainly consists of class and property hierarchies, and a dynamic part, which consists of class individuals and statements built from the individuals and properties.

In the proposed orchestration framework, the equipment model is maintained by a dedicated web service, which is entitled Ontology Service. As the internal states of the services change, the dynamic part of the semantic model is updated, whereas the static part consisting of class and property hierarchies remains unchanged. Generally, the equipment model is completely specific to the application scenario. While some parts of an equipment model might be reusable in several problem domains, it is unnecessary to define a concept ontology that would be applicable to several scenarios.

Since semantic models contain classes and properties as well as statements, they are effectively ontologies. In this dissertation, the word ontology is used as a synonym for semantic model. In addition to Ontology Service, the web service orchestration framework includes three other support web services: Ontology Manager, Orchestration Engine and Service Monitor. These four services and their interaction are described in the sequel.

Ontology Service maintains a semantic domain model expressed in OWL and provides a web service interface for query and update access to the model. The service interface includes several operations, of which Table 3.1 summarizes those that are used in this section, except for the *SetBaseOntology* operation, which will be summarized in Table 3.4.

The Ontology Service has the operation *ExecuteQuery*, which executes SPARQL SELECT queries [79] to obtain information on the semantic model, as well as the operation *ExecuteUpdate*, which executes SPARQL/Update [25] statements to modify the equipment ontology. In addition, Ontology Service includes the operation

Table 3.1: Ontology Service provides a web service interface for querying and manipulating the domain model.

Operation Name	**Inputs**	**Outputs**	**Purpose**
GetNewerOntology-Model	Model timestamp (integer)	• Model timestamp (integer) • Model download URL (string)	Determines if a newer domain model is available and from where it can be downloaded.
ExecuteQuery	SPARQL SELECT query (string)	Query result table (XML)	Evaluates a SELECT type query against the domain model.
ExecuteUpdate	A SPARQL/Update expression (string)	Result code ('SUCCESS' / 'FAILURE')	Updates the domain model with a single SPARQL/Update expression.
ExecuteSelectAnd-Pick	• A SPARQL SELECT query (string) • A query variable name (string) • Result table row index (non-negative integer)	Result value (string)	Executes a SELECT type query and picks the value of the specified variable from the result table.
FindShortestPath	• Start node URI (string) • Property URI (string) • End node URI (string)	A sequence of URIs	Finds the shortest path between two nodes along the given property.

SetBaseOntology, which sets the hosted semantic model, to which all queries and update statements will be applied. The latter operation is typically invoked near the beginning of a BPEL process, and the URL of the appropriate ontology is provided as an input parameter. Ontology Service loads the ontology from the specified URL and creates a copy of it. The hosted ontology is encapsulated by the service, and it can be accessed only through the *ExecuteQuery* and *ExecuteUpdate* operations. Because the format of query results is rather complex, the results of the *ExecuteQuery* operation are difficult to use directly, for example, in a BPEL process. Hence,

Ontology Service also includes two operations that return more refined data: *ExecuteSelectAndPick*, which executes a SPARQL SELECT query and retrieves a single value of the specified query variable, and *FindShortestPath*, which determines the shortest path between two individuals based on the specified property. The former operation returns a simple string value, while the latter returns an array of string values. Hence, the return values of both of the operations can easily be used in a BPEL process. Ontology Service also provides some variants of the aforementioned operations, but they are ignored in this dissertation.

The equipment ontology hosted by an Ontology Service must constantly be updated as the states of the web services evolve. Therefore, the domain web services must include notification operations through which they notify all interested parties of changes in their states. A dedicated web service, Ontology Manager, subscribes to receive notifications from the domain web services. Based on the received notifications, it sends update requests to the Ontology Service, so that the equipment ontology remains synchronized with the states of the domain web services. A compatible domain web service must send a notification whenever the internal state of the service changes.

All web services participating in the orchestration process are modeled using OWL-S [71] descriptions, which the Orchestration Engine service uses in determining the service most suitable for performing each task. While the semantic description of a web service may be either embedded into the WSDL document describing the service or available through a URI, the application experiments in Section 5.3 mainly use the former approach. Elements in the WSDL definition are linked to semantic concepts by inserting SAWSDL [20] annotations into the WSDL definition. In particular, WSDL port types are annotated with references to OWL-S Profiles and WSDL operations are annotated with references to OWL-S Processes as recommended by Martin et al. [54]. To initiate a production process, an engineer develops overall orchestration instructions in BPEL and submits them to Orchestration Engine to be executed. For this purpose, Orchestration Engine service includes the *SwitchProcessSemantic* operation.

Fortunately, several publicly available software tools facilitate the creation of OWL-S descriptions. For example, OWL-S API [72] includes a sample application which uses the API to generate OWL-S descriptions directly from a WSDL [8] definition of a service interface. Figure 3.1 contains a screenshot of the application. However, since plain WSDL is able to represent only the syntactic properties of a service interface, the generated OWL-S descriptions should typically be further edited to contain actual semantic information on the service. Although this semantic information can be added with graphical tools, such as Protégé [78], it is practically necessary to manually edit some parts of OWL-S descriptions. In particular, most

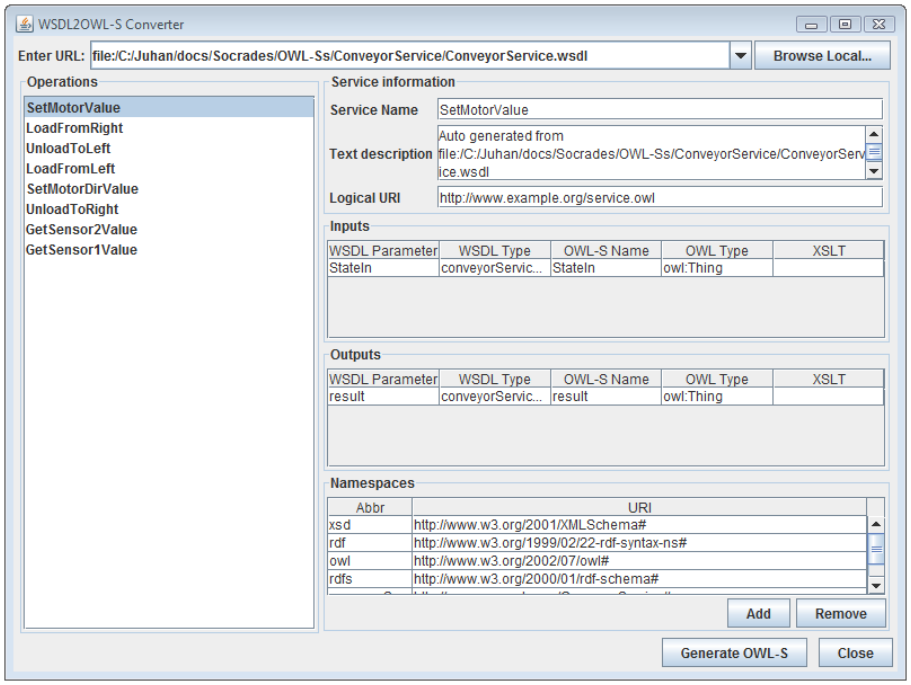


Figure 3.1: WSDL2OWLS, which is an example application included in the OWL-S API, generates OWL-S processes corresponding to operations defined in WSDL files.

graphical tools currently include poor support for the editing of condition and effect expressions.

The Orchestration Engine service is responsible for executing the BPEL processes, which are essentially instructions of how to orchestrate other web services so that they jointly achieve the overall task, such as producing a certain product. The partner links defined in the BPEL processes describe which web services are required for completing the processes. The partner link types are based on WSDL port types according to the BPEL standard, and the WSDL definitions refer to semantic concepts through SAWSDL annotations. Before starting to execute a process, Orchestration Engine determines a suitable web service for each uninitialized partner link. To find matching web services to be used in the partner links, Orchestration Engine depends on Service Monitor.

Service Monitor is a web service that detects and maintains a record of all available web services. It stores both the syntactic WSDL descriptions of the service interfaces and the semantic OWL descriptions. Service Monitor uses WS-Discovery [68] Hello and Bye messages to detect when web services become available or unavailable. In addition, it includes the *ScanNetwork* operation, invoking which causes Service Monitor to actively determine all available DPWS-compliant services.

Orchestration Engine invokes the *MatchService* operation of Service Monitor to determine the potential matches for each uninitialized partner link and selects the

most accurate match for each link. Service Monitor assigns a rating for each match indicating how accurately the candidate service matches the requested service. Since the requested service is indicated as a semantically annotated WSDL port type, the match rating is determined by comparing the semantic concepts referenced in the semantic annotations of the requested port type and the port type supported by the candidate service. A match rating of one indicates a perfect match, while zero indicates that the candidate service is unsuitable for use with the partner link.

Whether a web service is compatible with a certain partner link is determined by comparing the semantic concepts referenced by the WSDL definitions of the port type in the partner link specification and the port types of the candidate web service. Firstly, for a candidate port type to be compatible with a partner link port type, it must include a SAWSDL annotation referring to similar OWL-S Profiles. A candidate Profile matches exactly a request Profile, if it is the same OWL individual. The match is partial if the candidate Profile is a different instance of the request Profile class or one of its subclasses. In addition, the OWL-S Processes used in annotating the port type operations must match the Processes used in annotating the operations in the partner link port type. Service Monitor assumes a one to one mapping between WSDL operations and OWL-S Processes. Thus, if a WSDL operation is annotated with a list of references to several OWL-S Processes, only the first one is considered. Comparing two OWL-S Processes with each other involves comparing the input and output types as well as the preconditions and results. For example, the OWL-S API and its embedded reasoner can be used for determining whether the input of a candidate Process belongs to the same class as the input of the requested Process or to one of its sub classes. In the former case, Service Monitor awards a higher match rating to the two processes. The overall match rating between two OWL-S Processes is also affected by the semantic resemblance of the outputs, preconditions and effects of the Processes. The final match rating for a given service is computed as an average of the Profile and Process match ratings. However, if a single component evaluates to zero, then the overall match rating will also be zero. This can occur, for example, if no match can be found for a certain OWL-S Process.

Typically, one of the partner links in the executed BPEL process should represent the Ontology Service used in hosting the equipment ontology, based on which the current status of the production equipment is determined. The BPEL process may then contain an Invoke activity requesting Ontology Service to load a semantic model from a certain URL. The BPEL process should also specify the update rules for Ontology Manager to use in updating the semantic model. Furthermore, the BPEL process specifies what information is queried from Ontology Service and how that information is used in decision making. Nonetheless, the process contains only

standard BPEL constructs. Only the initial communication between Orchestration Engine and Service Monitor to resolve partner links is pre-programmed into the software components. This allows considerable flexibility in designing the orchestration instructions in BPEL. For example, BPEL processes, in general, are not required to use an Ontology Service or Ontology Manager. While Ontology Service and Ontology Manager can support the decision making in a BPEL process, the developer of the process has considerable freedom in defining the specifics of how these two support services are used.

Orchestration Engine uses the syntactic web service interface information specified in WSDL when invoking web service operations. Thus, only Service Monitor uses the semantic information in determining the compatible web services for each task.

The PC-based (executed on Personal Computer) web services described in this dissertation are compliant to the Devices Profile for Web Services, or DPWS [17], specification. The OASIS [67] consortium has standardized the DPWS specification version 1.1.

The web service interface of Ontology Manager includes several operations, of which Table 3.2 lists the most notable. The *ScanNetwork* operation causes Ontology Manager to detect all web services that may send event notifications and subscribe to receive the notification messages. When a notification message is sent by a service, Ontology Manager sends a request to Ontology Service to execute a SPARQL/Update statement which will update the ontology model according to the notification. In determining the SPARQL/Update statements to invoke, Ontology Manager relies on a set of update rules. The rules may be set by invoking the *SetUpdateRules* operation. Each update rule consists of a condition part and an action part. The condition part refers to the type and contents of notification messages and determines when the update rule should be applied. The condition part may also specify the event source by its endpoint URI or some other string identifier. The action part consists of a list of SPARQL/Update statements that are to be executed when the condition part matches a received notification message. The SPARQL/Update statements may actually be templates which contain keywords that refer to the content of the notification message. The keywords are expanded before the statements are executed. A keyword in a SPARQL/Update statement template is indicated by start and end delimiters, and it should match the local name of an element in a notification message. Each keyword is expanded by finding the first element in the notification message with the local name specified by the keyword and replacing the keyword, and the delimiters, with the text content of the element. However, this approach may produce undesired results when a notification message contains several elements sharing the same local name.

Table 3.2: Ontology Manager provides a web service interface through which it is possible to specify the ontology update rules.

Operation Name	**Inputs**	**Outputs**	**Purpose**
ScanNetwork	Subscription duration in milliseconds	The number of web services to which a subscription was made	Discovers web services and subscribes to receive event notifications from them.
SetUpdateRules	The update rules to use in XML form	A success code ('SUCCESS' / 'FAILURE')	Specifies the rules for updating the domain model.
SetEndpointMapping	A list of string ID - endpoint URI pairs	A success code ('SUCCESS' / 'FAILURE')	Specifies aliases to endpoint URIs.
SetActivityState	The new operation mode ('PASSIVE' / 'ACTIVE')	A success code ('SUCCESS' / 'FAILURE')	Selects between the rule editing and notification listening modes.

Especially if the orchestration scenario involves several web services of the same type, the update rules used by Ontology Manager should also specify the correct event sources. Otherwise, a similar event notification received from different service instances would cause the equipment model to be updated similarly. Service instances can be identified by their endpoint URIs. However, since Orchestration Engine may select any semantically compatible web service for any partner link, using an endpoint URI to specify an event source is infeasible. Instead, the update rules should specify the event sources using the correct partner link names. Orchestration Engine may then invoke the *SetEndpointMapping* operation of Ontology Manager to indicate the mapping from partner link names to endpoint URIs based on the results of semantic partner link resolution. That the update rules depend on the executed BPEL process poses no problem, since the update rules are typically specified in the BPEL process. It is, therefore, straightforward to modify both a BPEL process and the corresponding update rules simultaneously.

Typically, a BPEL process initializes the Ontology Manager service by invoking the operations *ScanNetwork* and *SetUpdateRules* near the beginning of the sequence. Then, it activates the update rules by invoking the *SetActivityState* operation. While the initialization may be alternatively performed through the graphical user interface of Ontology Manager, programming it into the BPEL process reduces the amount of

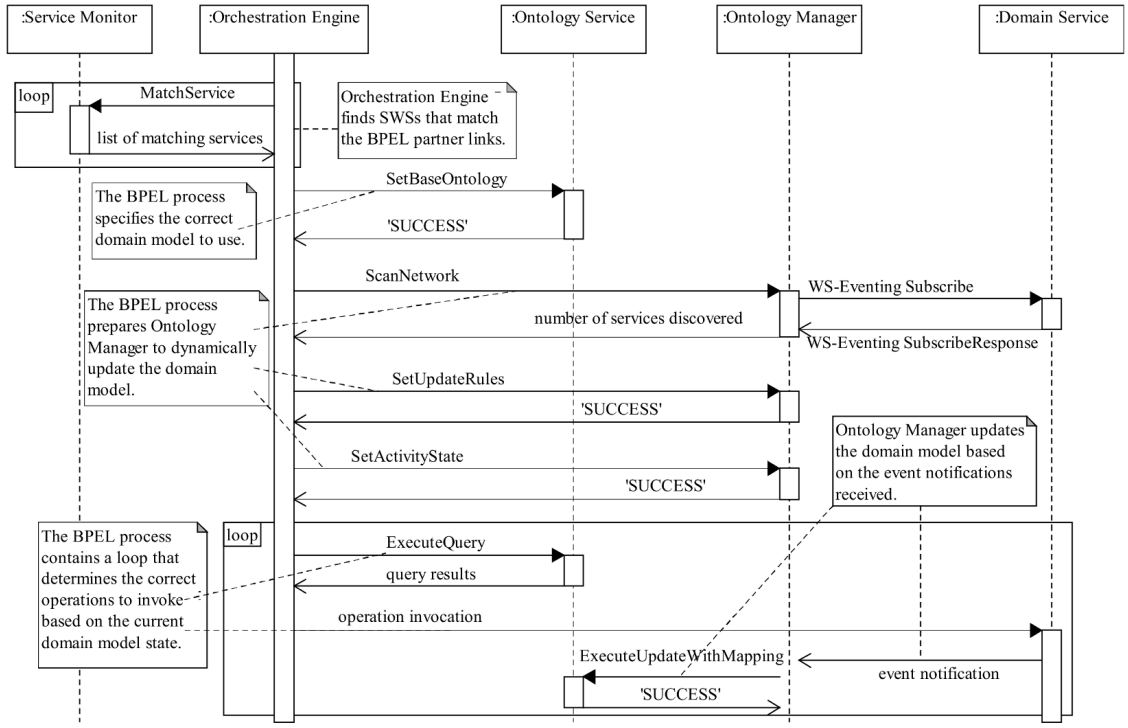


Figure 3.2: Ontology Service hosts the equipment ontology, which describes the states of the domain web services, and Service Monitor hosts the web service descriptions.

user interaction required. When the ontology update rules have been activated, and Ontology Manager receives a notification message, it compares the URI of the source web service and the contents of the message to the condition parts of all update rules. All conditional effects of the matching update rules are executed by invoking the operation *ExecuteUpdateWithMapping* of Ontology Service. Section 3.4.2 discusses the details of Ontology Manager and the dynamic domain model update approach.

The abovementioned software components can be executed in remote locations. The interaction of the software components is illustrated in the sequence diagram of Figure 3.2. For clarity, the domain web services are represented as a single object in the diagram. Orchestration Engine resolves the partner links before executing a BPEL process. The executed BPEL processes are not required to fully conform to the sequence in Figure 3.2, which only illustrates the most typical use case.

3.3 OWL-S and SPARQL-based Semantic Web Service Composition

The approach proposed in this section involves three web services with specialized functionality to control production systems. The services have the following names: Orchestration Engine, Ontology Service, and Service Monitor. Orchestration Engine

executes BPEL processes defining the desired production goals, Ontology Service maintains an OWL model of the current world state, and Service Monitor composes the available semantic web services to achieve the production goals considering the current world model. The three web services can be used in BPEL-based web service orchestration [80]. However, since BPEL operates primarily on syntactic web service descriptions, it is considerably less essential for the approach described in this section.

Similarly to [89], the approach proposed in this section uses OWL-S and SPARQL to semantically describe the available web services while using OWL for describing world states. In addition, the presented approach involves automatically updating the world model based on event notifications sent by the semantic web services, as described in Section 3.4.3.

3.3.1 Composition Pattern Overview

Figure 3.3 illustrates the interaction between Orchestration Engine, Ontology Service, and Service Monitor to compose domain web services. Orchestration Engine executes a BPEL process, which invokes the *FulfilGoal* operation of Service Monitor with the appropriate production goal and restriction expressions. This step is represented by interaction 1 in Figure 3.3. Service Monitor then invokes Ontology Service to determine the temporally accurate domain data (2). Based on the current domain state and the semantic domain web service descriptions, Service Monitor plans a sequence of atomic process executions to achieve the goal. If Service Monitor finds such a sequence, it generates and executes a corresponding OWL-S composite process. The composite process comprises an OWL-S *Sequence* element which contains *Perform* elements that sequentially execute the atomic processes with appropriate input values. As each atomic process is grounded to a WSDL operation, Service Monitor effectively invokes a sequence of domain web service operations (4). As the web services are invoked, their internal states change, causing them to send event notifications (5). Service Monitor analyzes the semantic descriptions of the notification operations, which are also included in the OWL-S documents, and sends the appropriate domain model update requests to Ontology Service (6). Hence, the domain model hosted on Ontology Service remains synchronized with the most recent production system state while Service Monitor fulfills a production goal. The latter step is described in detail in Section 3.4 and corresponds to the world state transition described in [89].

To support scenarios in which domain web services send no event notifications, Service Monitor has an optional mode that can be activated and deactivated by invoking the *SetParameter* operation with appropriate input parameters. By default, Service Monitor reacts to incoming notifications. However, when the mode is deac-

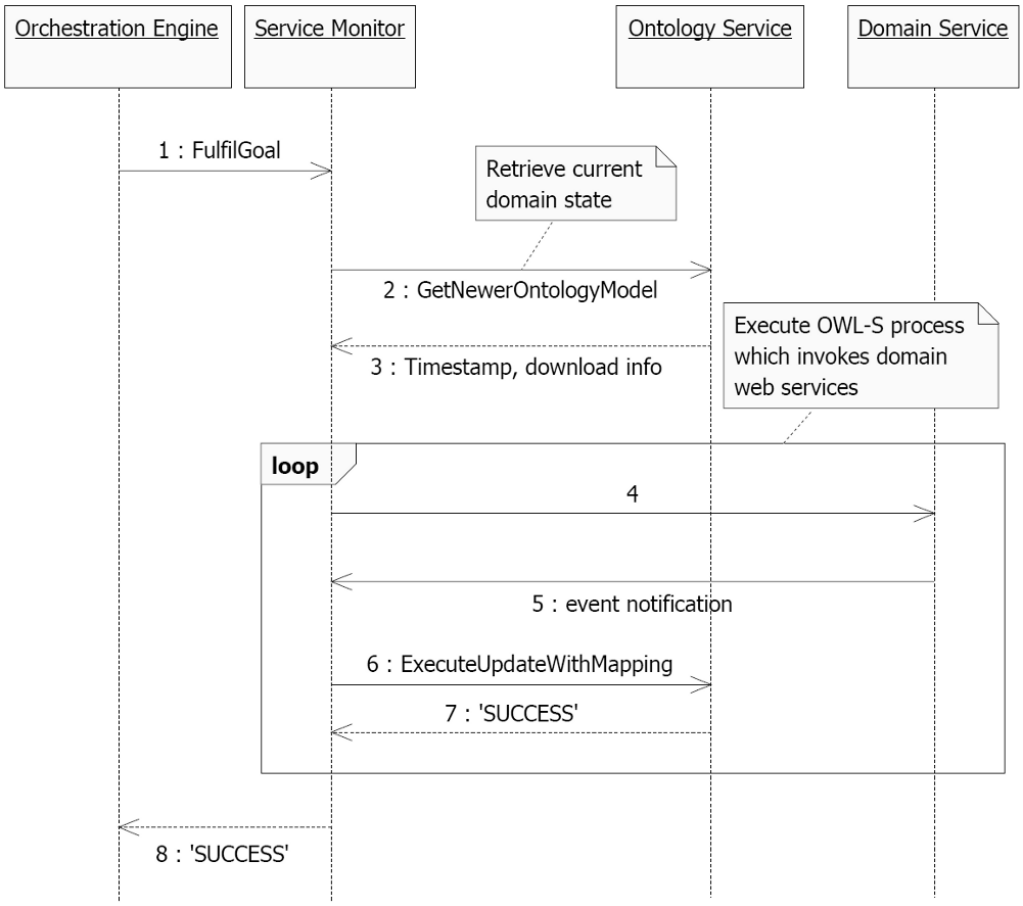


Figure 3.3: Orchestration Engine executes BPEL processes which request Service Monitor to orchestrate domain web services with the aid of Ontology Service.

tivated, Service Monitor ignores event notifications and, instead, each time Service Monitor executes an atomic process, it generates a SPARQL/Update statement to accordingly update the domain model and submits the statement to Ontology Service by invoking the *ExecuteUpdate* operation.

Ontology Service hosts a semantic model of the production system, which is called the domain ontology. To enable reliable decision-making, the domain model must remain synchronized with any changes in the world state. Therefore, Ontology Service provides the *ExecuteUpdate* and *ExecuteUpdateWithMapping* operations, which execute SPARQL/Update expressions to update the model. The latter operation additionally accepts a mapping for variable values and condition expressions for deciding which of the update expressions to execute. In addition, the *GetNewerOntologyModel* operation makes it possible to determine both the current time stamp of the domain model and a network address from which the entire model can be downloaded.

Service Monitor maintains a record of the OWL-S descriptions of all available semantic web services. The combined semantic service descriptions form the service

model. Service Monitor discovers available web services using WS-Discovery. After discovering a service, it retrieves the WSDL definition. Since WSDL can only describe the syntactic interface of a web service, an OWL-S document is needed to describe the service semantically. Therefore, Service Monitor searches each WSDL document for an embedded OWL-S document. If the actual OWL-S document is extensive, the embedded document may contain only a statement importing the main document from an external URL.

When Service Monitor is used in BPEL-based web service orchestration, its purpose is mainly to find semantic matches for web service interfaces appearing in BPEL process partner links [80], while Orchestration Engine executes the workflow described by the BPEL processes. Once Service Monitor determines a set of suitable web services, Orchestration Engine attempts to modify the BPEL process to refer to the new syntactic interfaces. However, since the SPARQL expressions in the OWL-S service condition and effect statements refer to instance data in the domain ontology, selecting an appropriate web service requires that the latest domain data is available. Moreover, calculating the semantic resemblance between two SPARQL expressions is problematic. The current version of Service Monitor therefore follows the common approach of ignoring condition and effect expressions when computing the semantic resemblance between two services. The new approach presented in this section solves the problem by automatically composing appropriate process prescriptions in OWL-S based on production goals formulated in SPARQL.

In the new approach, Service Monitor composes and executes semantic web services based on the goal information sent to it by Orchestration Engine. In particular, the BPEL processes executed by Orchestration Engine only specify the production goal information instead of the actual web service invocation workflow. However, Orchestration Engine still relies on Service Monitor in decision making as well as discovery of suitable services to be used as partner services in executing the BPEL processes.

A large set of Orchestration Engine service instances can exist, and the instances can be geographically distributed. For example, each production plant might have one or more Orchestration Engine services. A client can then select one of them to produce a product. Because the process instructions are expressed as a BPEL process and semantic information is used in expressing partner service requirements, each instance of the Orchestration Engine service can interpret the same instructions, and the exact type of hardware present at the corresponding production plant is irrelevant. If certain types of production equipment are unavailable at the selected site, the orchestration framework will fail to find suitable partner web services and will reject the process. In that case, the client may select another Orchestration Engine instance from another plant. Alternatively, a mediator service may be deployed

at each production site. Each mediator then coordinates the set of Orchestration Engines available at that site to respond to received production requests. However, this dissertation will focus on the functionality at the level of a single Orchestration Engine.

When Service Monitor is used for semantic web service composition, it analyzes the temporally accurate instance data as well as the service precondition and effect expressions to determine the applicability of services in achieving a specific production goal. However, since instance data is required, the service selection, or composition, must be performed immediately prior to attempting to achieve the goal [89].

Service Monitor provides the *FulfilGoal* operation for achieving production goals. When the operation is invoked, Service Monitor attempts to plan a process sequence for achieving the goal. Once Service Monitor has found a solution plan, it generates the corresponding OWL-S composite process description, stores it in its internal OWL-S knowledge base, and executes the composite process. In addition to a single SPARQL ASK query representing the goal, the *FulfilGoal* operation accepts additional parameters for guiding the planning process. The additional parameters are discussed in the following paragraphs, and they should reduce the number of different system state trajectories that Service Monitor will have to consider. Without such a reduction, the search algorithm discussed in Section 3.3.2 may exhibit poor performance and deplete the memory resources available.

Firstly, the service client may break down a complex goal into a set of sufficiently simple subgoals. Hence, the inputs comprise an array of SPARQL queries representing the subgoals that must be fulfilled in order to achieve the final goal, which is represented by the last array element. Provided that the client is able to break down the goal into sufficiently simple subgoals, Service Monitor can sequentially solve the subgoals instead of attempting to find a solution plan achieving the final goal at once.

Finally, the *FulfilGoal* operation accepts as input a SPARQL ASK query representing a restriction that all the intermediate domain states must satisfy. This allows the planning algorithm to ignore any system trajectories traversing the disallowed domain states. Table 3.3 summarizes those operations in the Service Monitor service interface that are used in this section.

In summary, the approach presented in this section involves formulating a BPEL process that invokes the *FulfilGoal* operation of Service Monitor with a SPARQL ASK query stating the production goal. In the simplest case, the BPEL process contains only a single invocation of the *FulfilGoal* operation preceded by the assignment of the BPEL variables with the appropriate queries. Alternatively, the process may contain more complex logic, such as a *ForEach* loop repeating a sequence of

Table 3.3: Service Monitor provides a web service interface that allows the registering of new production goals.

Operation Name	**Inputs**	**Outputs**	**Purpose**
ScanNetwork	'CLEAR' / 'RETAIN'	Number of semantic web services discovered	Discovers semantic web services in the local network.
SetParameter	• Parameter type • Value ('true' / 'false')	Response code ('SUCCESS' / 'FAILURE')	Disables or enables a feature specified by the parameter type.
FulfilGoal	• A list of goals (SPARQL ASK queries) • A state-space restriction (SPARQL ASK query)	Response code ('SUCCESS' / 'FAILURE' / 'IMPOSSIBLE' / 'BUSY')	Attempts to synchronously fulfil a goal.

FulfilGoal operation invocations. When Orchestration Engine executes the BPEL process, it effectively requests Service Monitor to fulfill the production goals.

3.3.2 Service Composition Algorithm

When determining a service composition capable of accomplishing a production goal, Service Monitor analyzes the changes different OWL-S processes would cause in the domain state. As the search progresses, the algorithm builds a state-space graph according to Algorithm 1. Following [89], the nodes of the graph are domain states, each of which is represented by an RDF graph containing the statements that are valid in the state. A directed arc between two nodes represents an OWL-S process with a certain combination of input values causing a change in the domain model and hence a transition to a new state-space node.

Algorithm 1 is based on Algorithm 1 in [89]. However, the algorithm presented in this section is different in that it also considers conditional process effects, which in OWL-S are called *results*. In addition, while the algorithm of [89] considers process preconditions and effects to be grouped in a single CONSTRUCT query, Algorithm 1 first evaluates the preconditions (ASK queries) and then proceeds to evaluate the conditions (ASK queries) of each process result. Only if both the preconditions of the process and the conditions of the result evaluate to true does the algorithm evaluate the result effect expressions (SPARQL/Update statements).

The main difference between Algorithm 1 and the algorithm in [89] is that while the latter selects all operation invocations that directly lead from the current state

Algorithm 1 Compute the path to a goal state.

Require: $StartNode.depth \geq 0 \wedge maxDepth \geq 0$
  $Unprocessed \Leftarrow \{StartNode\}$
  while $Unprocessed \neq \emptyset$ **do**
    choose the first node $Current \in Unprocessed$
    $Unprocessed \Leftarrow Unprocessed \setminus \{Current\}$
    $depth \Leftarrow Current.depth + 1$
    if $depth > maxDepth$ **then**
      return failure
    end if
    $domain \Leftarrow Current.state$
    for all $process$ in $Processes$ **do**
      if $!evaluate(process.preconditions, domain)$ **then**
        continue
      end if
      for all $result$ in $process.results$ **do**
        $effects \Leftarrow result.effects$
        $conditions \Leftarrow result.conditions$
        $value \Leftarrow evaluate(conditions, domain)$
        if $effects = \emptyset \vee !value$ **then**
          continue
        end if
        $solutions \Leftarrow findSolutions(domain, conditions)$
        for all $solution \in solutions$ **do**
          $state \Leftarrow compute(domain, effects, solution)$
          $Next.state \Leftarrow state$
          $Next.depth \Leftarrow depth$
          $Edge \Leftarrow (Current, Next, process, solution)$
          $Graph.nodes \Leftarrow Graph.nodes \cup \{Next\}$
          $Graph.edges \Leftarrow Graph.edges \cup \{Edge\}$
          if $Next.state$ satisfies the goal **then**
            return $Next$
          end if
          $Unprocessed \Leftarrow Unprocessed \cup \{Next\}$
        end for
      end for
    end for
  end while

to the goal state, the former attempts to find the minimal sequence of operation invocations that leads from the current world state to the goal state.

Because Algorithm 1 analyzes all system trajectories that are possible with the set of OWL-S processes available, the state-space expands rapidly as the algorithm proceeds to new levels. In each domain state, the algorithm considers each applicable OWL-S process result. In addition, since different input values may cause

different changes in the domain state, the search algorithm considers all potential combinations of input values for each result.

The search algorithm is based on breadth-first search and terminates after encountering a domain state satisfying the goal. Hence, the algorithm evaluates the ASK query representing the goal in each encountered domain state. However, to conserve memory and accelerate node comparison, the state-space graph nodes contain only the differences in the associated domain states compared to the initial state.

Using breadth-first search instead of depth-first search makes it possible to expand the search graph one level at a time, which guarantees that any solution plan discovered is minimal. However, to avoid exceeding memory limits, the search algorithm must contain an additional termination condition. For example, Algorithm 1 uses the maximum considered plan length, denoted by *maxDepth*, to determine when to discontinue the search.

Both the memory requirements and computational complexity of Algorithm 1 are exponential in the number of OWL-S processes available. This is because the algorithm will analyze the entire state space until finding a state satisfying the goal. In particular, it does not prune branches of the state space that cannot lead to a goal state. Nonetheless, because of the elaborate computation and evaluation of each condition and effect expression, provided infinite memory and computational resources, the planner will eventually find the minimal solution plan. Consequently, the search algorithm is both sound and complete.

Despite the exponential complexity, Algorithm 1 is practically applicable, provided that the goal expression is accompanied by a restriction expression that allows the planner to focus on only the state space trajectories that lead to the goal state. The restriction expression is provided as an optional argument to the Service Monitor *FulfilGoal* operation.

3.4 Event-based Updating of a Domain Model

The web service-based domain model hosting and updating approach applied in this section was originally presented in [80]. It has later been somewhat refined, and the current status in presented in Section 3.4.2. An alternative approach was presented in [84], and its current status is briefly presented in Section 3.4.3. The main principle in both approach variants is that a web service, *Ontology Service*, hosts a domain model that is formulated in OWL and describes the current world state. *Ontology Service* is described in Section 3.4.1.

3.4.1 Ontology Service

Ontology Service allows query and update access to the hosted OWL model through a web service interface. Queries sent to Ontology Service follow the SPARQL [79] syntax, and update requests are formulated in the SPARQL/Update [25] language. When Ontology Service is started, it initially hosts no OWL model. Therefore, it is necessary to invoke the *SetBaseOntology* operation. The single parameter to the operation is the URL specifying the internet address from which Ontology Service may read the domain description. Once the domain model has been set, it is possible to evaluate SPARQL ASK queries against the model by invoking the *ExecuteAskWithMapping* operation and modify the model with SPARQL/Update expressions by invoking the *ExecuteUpdateWithMapping* operation. Ontology Service additionally provides other operations that, for example, execute SPARQL SELECT queries. Because only the aforementioned three operations are relevant for the approach presented in this section, Table 3.4 omits the operations that are used in other sections of this dissertation.

The input data for the *ExecuteAskWithMapping* operation can be a simple SPARQL

Table 3.4: Ontology Service provides a web service interface for querying and manipulating the domain model.

Operation Name	**Inputs**	**Outputs**	**Purpose**
SetBaseOntology	OWL document URL	Result code ('SUCCESS' / 'FAILURE')	Reads in a domain description and replaces the previous model.
ExecuteAskWithMapping	• a SPARQL ASK-type query • a variable value mapping (optional)	Query result ('yes' / 'no')	Evaluates a SPARQL ASK-type query against the current domain model.
ExecuteUpdateWithMapping	• A list of conditional effects (SPARQL ASK-type queries, SPARQL/Update expressions) • A variable value mapping (optional)	Result code ('SUCCESS' / 'FAILURE')	Updates the domain model with the effect expressions whose conditions evaluate to true.

ASK query. Alternatively, the query can be customized by additionally providing a value binding to one or more variables occurring in the query. The value binding is provided in the same format as for the *ExecuteUpdateWithMapping* operation, which has a somewhat more complex input data XML structure due to the larger variety of data elements. The UML class diagram in Figure 3.4 illustrates the input data XML structure for the latter operation. The operation accepts conditional update expression sets. Ontology Service will then evaluate the conditions against the current domain model, and only if the conditions evaluate to true, Ontology Service will update the domain model by executing the update expressions.

Another web service, *Ontology Manager*, selects an available Ontology Service and sends update requests to it based on the received event notifications and a set of user-specified update rules. The approach has later been refined so that a more advanced web service, *Service Monitor*, replaces *Ontology Manager*. Service Monitor retrieves the appropriate update requests from the semantic descriptions of the web services [84]. Thus, no extra information, such as update rules, needs to be provided.

Figure 3.5 illustrates the role of the event listener service. The following subsections discuss both of the approach variants, in which the event listener service is either Ontology Manager or Service Monitor.

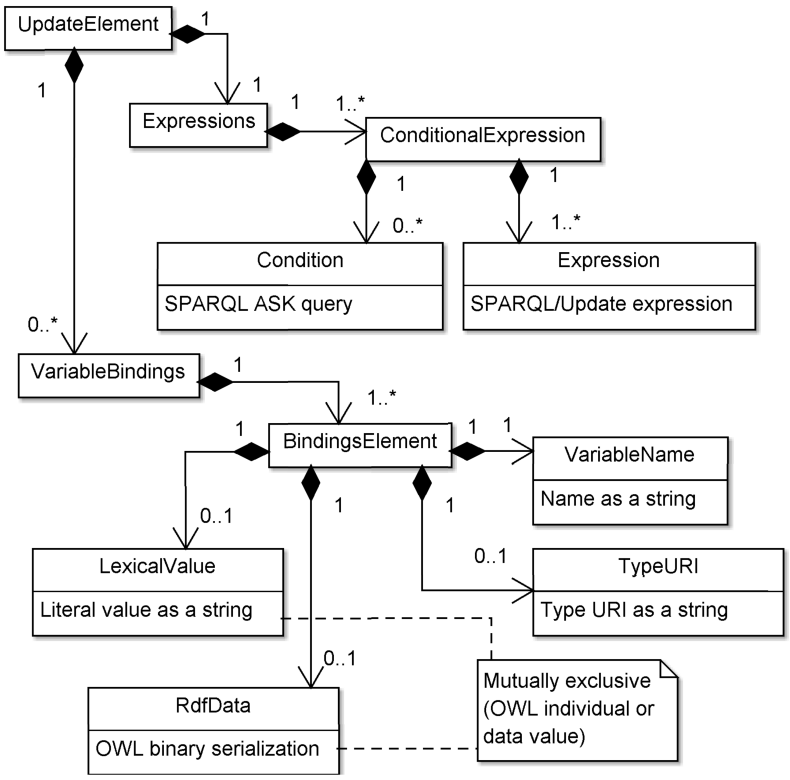


Figure 3.4: The Ontology Service *ExecuteUpdateWithMapping* operation has a complex input message XML structure.

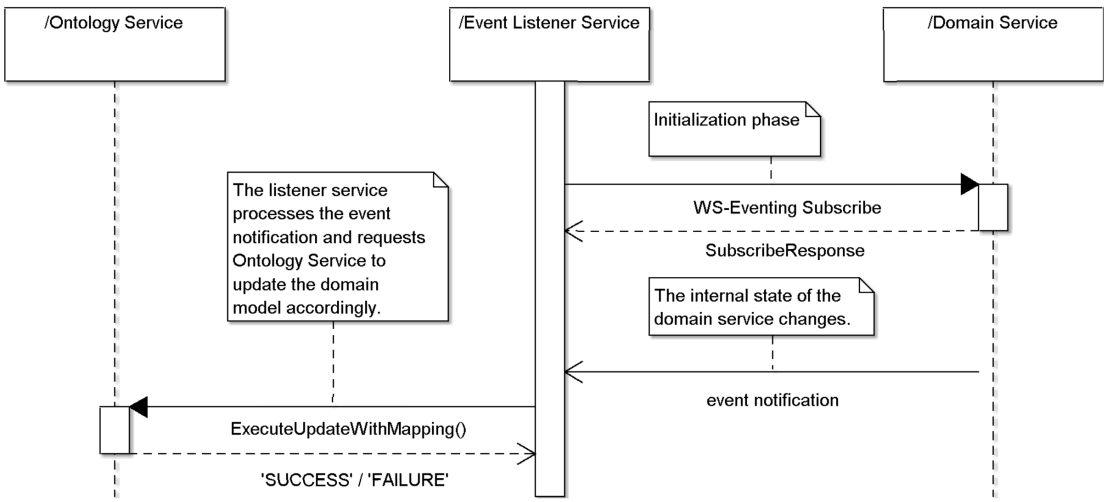


Figure 3.5: An event listener service translates event notifications from domain web services to appropriate updates to the domain OWL model.

3.4.2 The Ontology Manager Approach

Ontology Manager is a web service that connects with an Ontology Service to update the domain model based on event notifications.

Ontology Manager provides all of its services through the web service interface outlined in Table 3.2. The service interface enables remote software actors to interact with Ontology Manager. In addition, Ontology Manager includes a graphical user interface (GUI) for direct user interaction. The GUI is particularly useful for creating update rules and is accessible through the Ontology Manager host computer.

To receive event notifications from domain web services, Ontology Manager must subscribe to them. To this end, the Ontology Manager web service interface includes the *ScanNetwork* operation. Invoking the operation causes Ontology Manager to discover all available web services using the Web Services Dynamic Discovery (WS-Discovery) [68] protocol. After discovering a service, Ontology Manager subscribes to receive all event notifications from it using the Web Services Eventing (WS-Eventing) [119] protocol. The *ScanNetwork* operation can also be invoked through the GUI menu bar.

Whenever Ontology Manager receives an event notification, it consults a set of update rules to determine the appropriate update request to send to Ontology Service. An update rule essentially consists of a set of conditions and a set of effects to be applied when the conditions match an incoming event notification. Ontology Manager compares the conditions directly to the received event notification, and if they match, applies the effects by invoking the Ontology Service *ExecuteUpdateWithMapping* operation.

Update rule conditions include the notification source endpoint URI, the WSDL

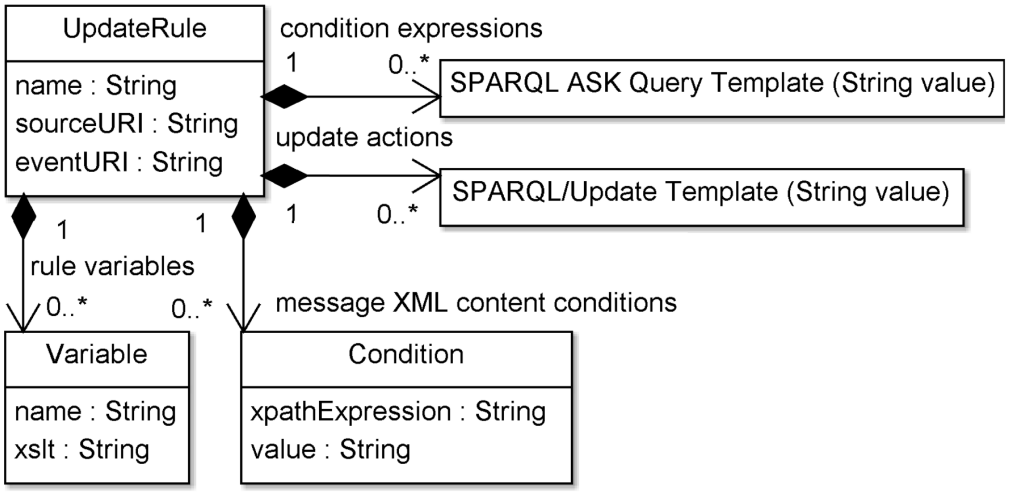


Figure 3.6: The conditions in Ontology Manager update rules directly refer to notification message contents.

notification operation action URI and the conditions on notification message content. The latter conditions are a list of pairs formed by an XPath [10] expression and a string value. The XPath expression identifies an XML element or element attribute in a notification message, and the string value specifies the required text content of the element or the attribute value.

The update rule effects are a list of SPARQL/Update expression templates. The templates may also contain keywords indicated by enclosing them within '#$' and '$#'. The keywords must be XML element names. Before sending the update expressions to Ontology Service, Ontology Manager will replace each keyword with the text content of the corresponding XML element in the received notification message, so that the templates are expanded into valid SPARQL/Update expressions. In addition, update rules may contain variables. Each variable has the same name as some SPARQL query variable occurring in the update expressions and an XSLT [9] script that extracts the variable value from the received notification message. The variable value mapping will then be included in the request message when Ontology Manager invokes the *ExecuteUpdateWithMapping* operation. Figure 3.6 illustrates the update rule structure.

The update rule conditions refer directly to the XML structure of the notification messages. Hence, they require stable service interfaces. In particular, if the XML Schema structure of a notification message is altered, the update rules referring to that notification will most probably cease to function as intended. Therefore, an alternative update rule type called *semantic update rules* was proposed in [83]. However, such an approach would require difficult rules on applying semantic annotations and compromise the complete independence of the Ontology Manager

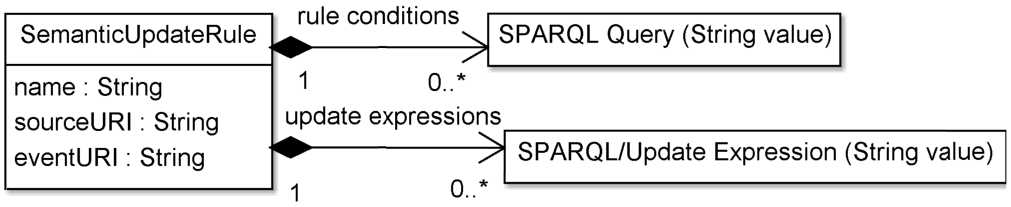


Figure 3.7: Rule conditions in the abandoned semantic update rule approach referred to OWL models through SPARQL expressions.

approach from the service descriptions. Moreover, the only benefit of the alternative approach, the ability to apply the same update rules regardless of modifications to the service interface, would be merely artificial as it would only shift the effort into the development of the service descriptions. Therefore, the alternative approach has been abandoned in the current version of Ontology Manager, and any XSLT transformation scripts must now be specified in the rules themselves as described above.

Semantic update rules differ from the conventional update rules in that their conditions do not refer directly to the notification message XML contents, but to an OWL model generated from the message by applying an XSLT script. The XSLT script must be referenced by the corresponding WSDL message part through an SAWSDL (Semantic Annotations for WSDL and XML Schema) [20] *liftingSchemaMapping* annotation. The condition list of a semantic update rule is a list of SPARQL ASK type queries to be evaluated against the OWL model resulting from the transformation. Ontology Manager would evaluate the condition expressions to determine if a rule matches. Figure 3.7 illustrates the structure of semantic update rules.

Whenever Ontology Manager receives an event notification, it compares the notification message to each update rule and executes each matching rule by invoking the *ExecuteUpdateWithMapping* operation with appropriate parameters.

Instead of notification operations, update rules may alternatively refer to request-response type operations. Ontology Manager will then invoke those operations periodically to determine when the rules should be applied. This approach is useful when a web service provides no event notifications but includes adequate request-response type operations for accurately determining the service status. The time interval of successive request-response operation invocations can be configured through the Ontology Manager GUI.

3.4.3 The Service Monitor Approach

Service Monitor is a web service that maintains a semantic web service database. It discovers semantic web services and stores their OWL-S descriptions. Then, when

the *FulfilGoal* operation is invoked, Service Monitor attempts to compose the OWL-S processes and executes the composition to fulfill a production goal [84]. Similarly to Ontology Manager, Service Monitor provides both a graphical user interface and a web service interface. The former interface facilitates direct user interaction, while the latter allows Service Monitor to act as a support service for other software-based actors. In addition, Service Monitor subscribes to event notifications from semantic web services, so that it may send update requests to Ontology Service.

After detecting a semantic web service, Service Monitor automatically reads the service WSDL document and the potential OWL-S description of the service. If the WSDL operations have SAWSDL annotations referring SWRL rules, Service Monitor generates the OWL-S processes based on the SAWSDL annotations in the WSDL document. Finally, Service Monitor extracts semantic update rule object models from the generated OWL-S descriptions.

The OWL-S derivation approach is based on SAWSDL annotations. In particular, Service Monitor generates one OWL-S process for each WSDL operation, and one OWL-S Process Result for each SWRL rule in the SAWSDL annotations of the operation. The result conditions and effects are extracted from the referenced SWRL rule. Furthermore, Service Monitor can be configured to generate the condition and effect expressions using either the SPARQL or SWRL expression language. The details of the OWL-S derivation process and the associated SAWSDL annotation conventions are described in Section 3.5.

Similarly to Ontology Manager, Service Monitor also provides a *ScanNetwork* operation for discovering all available web services, and Service Monitor connects to an Ontology Service instance, which it attempts to maintain synchronized with the most current domain state. In addition to subscribing to event notifications from web services, Service Monitor creates an object model quite similar to the Ontology Manager update rules from those OWL-S processes corresponding to WSDL notification operations. In OWL-S, an atomic process that has only output parameters corresponds most closely to a WSDL notification operation [54]. Therefore, Service Monitor converts each such process into a set of semantic notification objects. While Service Monitor could simply use the OWL ontology model containing the OWL-S processes, querying such an ontology at run-time would be slower than accessing the Java object model. Figure 3.8 shows the structure of the extracted notification objects next to the corresponding OWL-S process ontology concepts.

Each semantic notification object groups together a set of conditions and a set of effects that are expected to realize when the conditions hold. The conditions and effects are copied from the source OWL-S process results. In addition, a notification object has a set of notification variables that correspond to the outputs of the OWL-S process. Each variable has a name, type URI, and an XSLT script that

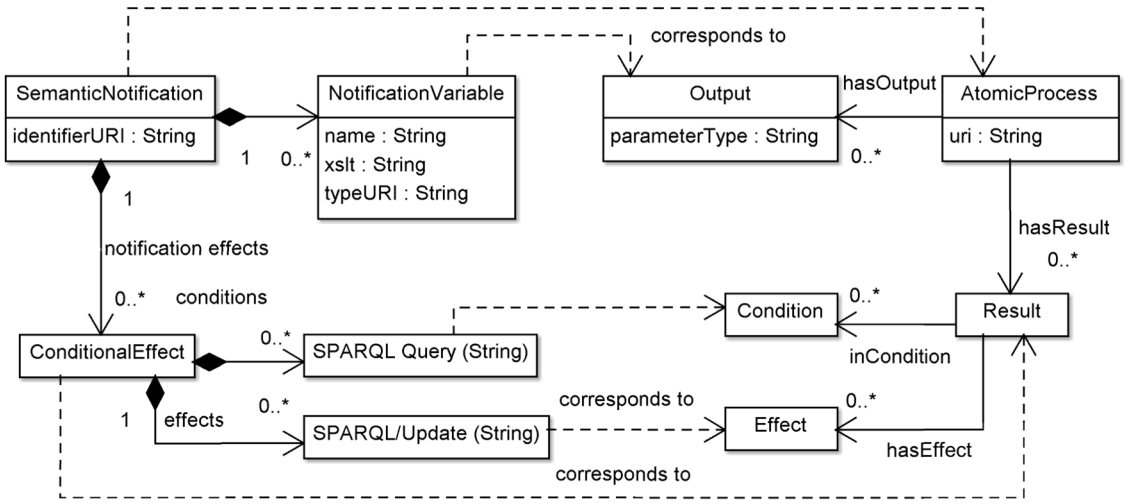


Figure 3.8: Service Monitor extracts a notification object model from OWL-S processes representing event notifications.

specifies how to obtain the variable value from the XML content of the notification message. The type URI is copied directly from the *parameterType* property value of the corresponding OWL-S process output. The XSLT scripts are extracted from the OWL-S atomic process groundings.

Each semantic notification object additionally includes an identifier URI, which is a copy of the OWL-S process URI. The identifier URI is used in identifying those semantic notification objects that may be discarded when a service leaves the network.

Ontology Service can only operate on SPARQL expressions. Therefore, the condition expressions in the update rules are SPARQL queries, and the effect expressions are SPARQL/Update statements. Thus, if the OWL-S model uses SPARQL and SPARQL/Update to represent condition and effect expressions, the expressions may be copied directly into the generated rule objects. However, as SWRL is also a popular expression language and integrates well with OWL due to its XML syntax, Service Monitor contains an adapter component that translates SWRL condition expressions to SPARQL ASK queries and SWRL effect expressions to SPARQL/Update expressions, which Service Monitor then copies into the rule objects generated. Since SWRL property atoms correspond quite directly to SPARQL subject-predicate-object triples, the conversion process is quite straightforward. Table 3.5 summarizes the conversion rules.

Basic SWRL can only express positive conditions and effects, which only require triples to be present in the model or add new triples into the model. Nevertheless, SWRL can be extended by defining new built-ins. Hence, negative antecedents and consequents can be formulated after defining custom built-ins called *noValue* and *remove*. The *noValue* built-in can occur in both conditions and effects, where

Table 3.5: SWRL atoms are converted to SPARQL triples and FILTER patterns.

SWRL Entity	**SPARQL Conversion**
Individual property atom *predicate(subject, object)*	Triple *subject predicate object*
Class atom *className(argument)*	Triple *argument a className*
Same individual atom *sameIndividual(arg1, arg2)*	FILTER pattern *FILTER* {*arg1 = arg2*}
Different individuals atom *differentIndividuals(arg1, arg2)*	FILTER pattern *FILTER* {*arg1 != arg2*}
Condition atom using the *noValue* builtin *noValue(subject, predicate)* *noValue(subject, predicate, subject)*	FILTER pattern *FILTER NOT EXISTS* {*subject predicate ?var*} *FILTER NOT EXISTS* {*subject predicate object*}
Effect atom using the *noValue* builtin	Triple in the DELETE pattern
Effect atom using the *remove* builtin *remove(0)*	Triple in the DELETE pattern *(The triple converted from the first condition atom)*

it either requires that there are no statements matching the argument pattern in the domain model or deletes the matching statements from the model, respectively. The *remove* built-in can only occur in effects and deletes the statements that caused the condition atom at the argument index to match. Both of the aforementioned built-ins are custom SWRL extensions inspired by the descriptions in [37].

Even when the OWL-S descriptions use SPARQL as the expression language for conditions and effects, the expressions may require some processing before they can be inserted into the update rule objects. In particular, if the effect expressions are SPARQL ASK queries, Service Monitor converts them into equivalent SPARQL/Update expressions, which are suitable for update rule effect expressions.

When Service Monitor receives a notification from a domain web service, it retrieves the corresponding semantic notification object based on the service endpoint URI and notification action path. Then, it obtains the values of the notification variables by applying their XSTL scripts to the received notification message. Service Monitor inserts the variable names and their values into the *ExecuteUpdateWithMapping* request message sent to Ontology Service. In addition, Service Monitor copies all of the conditional effects from the semantic notification object to the corresponding slots in the request message structure shown in Figure 3.4. When Ontology Service processes the request, it evaluates each set of condition expres-

sions with the specified variable values and executes each set of effect expressions for which the corresponding condition sets evaluate to true. Hence, depending on the current domain model state, a rule may cause no changes in the domain model even if the notification message matches the rule conditions.

While event notifications allow the passive reception of domain status updates, request-response type operations make it possible to actively poll the status of the services. Therefore, Service Monitor has recently been updated to extract similar object models from request-response type operations. However, to receive the response messages, Service Monitor must actively invoke the operations when it seems probable that the domain model is inaccurate.

3.4.4 Comparison of the Approach Variants

Both Service Monitor and Ontology Manager send the update requests to Ontology Service by invoking the *ExecuteUpdateWithMapping* operation upon receiving an event notification. While Ontology Manager requires a set of user-specified update rules to determine the appropriate updates to perform, Service Monitor extracts the update rules directly from the semantic descriptions of the web services. Thus, the Ontology Manager approach requires no semantically described web services; it only requires the development of a set of update rules based on the WSDL files describing the services. Table 3.6 summarizes the main differences and similarities between the two approaches.

While Service Monitor extracts the update rules automatically from the semantic descriptions of the web services, the approach requires quite extensive semantic service descriptions. For example, source data required for the OWL-S derivation process include SWRL rules in the domain model TBox, as well as semantic annotations in the service WSDL files referring to the appropriate TBox and ABox concepts.

On the one hand, the Ontology Manager approach places no constraints on the domain OWL model used, as it is typically possible to modify the update rules to accommodate a different OWL model. On the other hand, developing the update rules manually is laborious and more error-prone than the automatic rule derivation

Table 3.6: The domain model update approaches differ in the required source data.

Software Actor	**Update Trigger**	**Required Data**
Ontology Manager (web service)	Event notifications from domain web services	Update rules (SPARQL)
Service Monitor (web service)	Event notifications from domain web services	Semantic web service descriptions

performed by Service Monitor.

The update rules used by Ontology Manager may specify restrictions on the message content, which are then considered before applying the rule and sending the request to Ontology Service. However, message content restrictions are absent in the update rules used by Service Monitor, and all rule conditions are evaluated by Ontology Service. Therefore, the latter approach may result in more network traffic between Ontology Service and the event listener service.

3.5 Generating OWL-S from WSDL Documents

A WSDL document typically contains all of the information necessary to invoke a web service. Therefore, executable OWL-S processes may be derived from WSDL documents. However, the WSDL documents must contain SAWSDL annotations from which it is possible to determine the semantic details of the OWL-S processes, such as input parameter types as well as the conditions and effects of invoking the services. The approach presented in this section includes the typical method of annotating XML schema elements with references to OWL classes and providing XSLT transformations for translation between the XML schema and OWL. Additionally, this section proposes several additional conventions for use in the derivation of OWL-S descriptions from annotated WSDL documents.

Figure 3.9 illustrates the use of SAWSDL annotations in the proposed approach. In the figure, diamond shapes represent aggregation relationships, dashed arrows represent SAWSDL model references, and solid arrows represent other language-dependent references, such as references to variables in SWRL rules. The model references refer to OWL individuals by their URIs. XSLT transformation scripts have no URIs. Nonetheless, they can be saved as XML files and made accessible through URLs, so that the SAWSDL lowering and lifting schema mapping annotations can refer to the URLs.

SAWSDL sets no restrictions on how the actual OWL descriptions are accessed. In Figure 3.9, the WSDL document contains a small embedded OWL document which imports the actual OWL descriptions, such as the domain ontology and the referenced SWRL rules. In addition, the embedded OWL document may contain the *Binding* instances to be used in replacing variables in the SWRL rule referents in order to specialize the rule conditions and effects for the service instance described.

The software component that processes the service WSDL descriptions uses the SAWSDL annotations in generating executable OWL-S processes, namely instances of the class *AtomicProcess* defined in the OWL-S Process ontology. In the semantic web service orchestration framework described in this dissertation, the software component is a web service called Service Monitor [84]. Therefore, in the sequel, the software component will be referred to as Service Monitor. To allow the generated

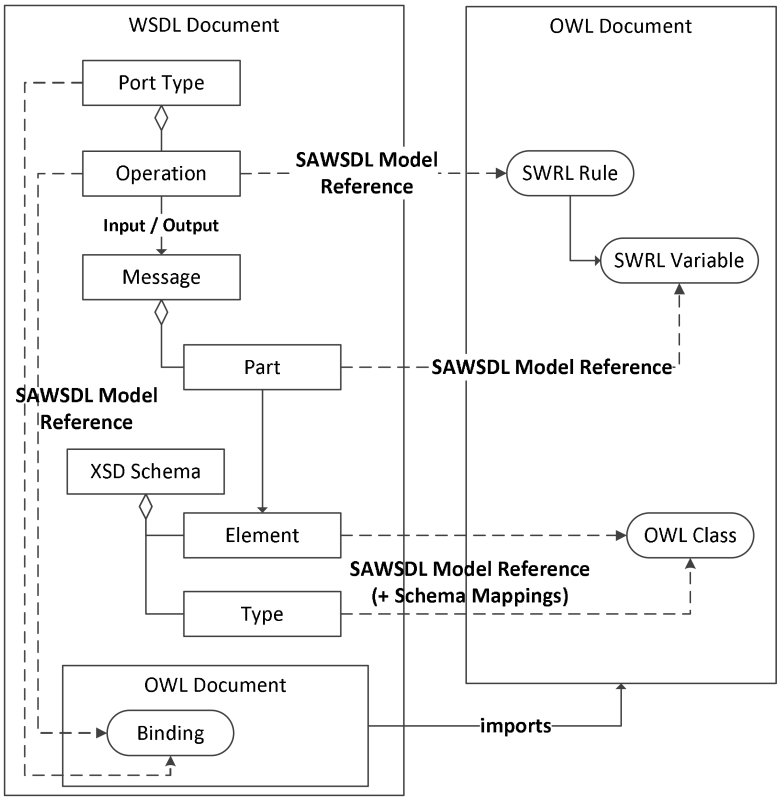


Figure 3.9: SAWSDL annotations are added to WSDL operations, port types, message parts and schema elements.

processes to be executed, Service Monitor additionally generates a *grounding* for each of the generated processes.

The UML class diagram in Figure 3.10 illustrates the structure of the OWL-S documents that Service Monitor generates from the annotated WSDL documents. Figure 3.10 represents the structure as a simplified UML class diagram, in which the classes correspond to the classes in the OWL-S ontology for which Service Monitor creates instances, and the associations represent the same-named OWL object properties in the OWL-S ontology. The attributes of the classes in the diagram represent the OWL datatype properties for which Service Monitor writes values.

The following subsections describe the proposed approach of applying SAWSDL annotations to specific WSDL entity types. The approach differs from the guidance given in the SAWSDL specification [20], which appears to provide suggestions rather than strict definitions [54].

3.5.1 WSDL Operations

The model references of WSDL operations may refer to SWRL rules and instances of the *Binding* class defined in the OWL-S Process ontology. Each rule is the source of the condition and effect expression extracted for one generated OWL-S Process

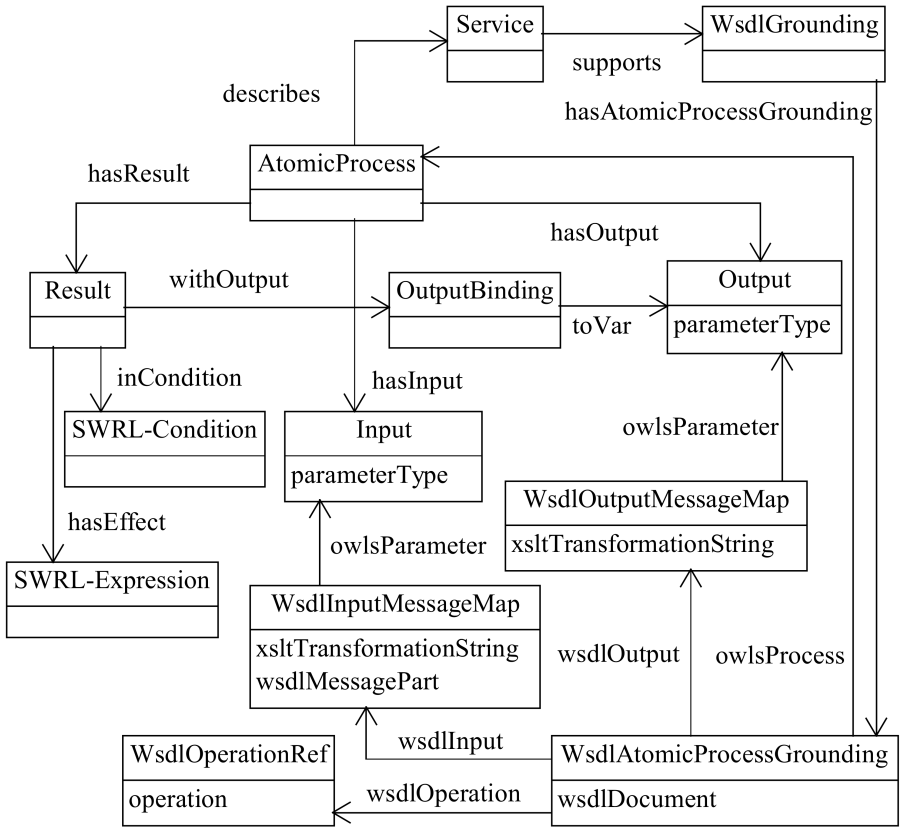


Figure 3.10: The generated OWL-S model both describes the service functionality and provides sufficient data to invoke the service.

Result. Each binding specifies an OWL individual to replace a SWRL variable when copying the conditions and effects from the rule references to the generated OWL-S Process Results. Thus, it is possible to formulate the conditions and effects for a group of web services of the same type using only a few SWRL rules.

As noted in Section 3.4.3, the SWRL rule syntax can be extended to also allow negative conditions and effects.

3.5.2 WSDL Port Types

The model references of a WSDL port type may refer to instances of the *Binding* class. Such references are a shorthand for annotating each of the port type operations with the *Binding* instances.

3.5.3 WSDL Outputs

The model reference of an Output may refer to an OWL individual. Each referent is interpreted to represent the value of the output after a successfull service invocation. Service Monitor will generate a corresponding *OutputBinding* for the generated *Process*.

3.5.4 WSDL Message Parts

Service Monitor will generate an OWL-S Process Input or Output for each WSDL message part used in an operation input or output. The model reference of the message part may specify the SWRL variable that the generated parameter will replace in the generated condition and effect expressions.

3.5.5 XML Schema Definitions

The model reference of a schema element may refer to an OWL class or to an OWL datatype property. Service Monitor uses the referents in determining the parameter types of the generated process parameters. The latter referent type is a special case, in which the parameter type is the range of the referent property. Hence, a datatype property referent indicates that the generated OWL-S parameter takes data values instead of OWL individuals.

Regardless of the referent type, lifting schema mappings may additionally be specified. Service Monitor will then download the XSLT scripts from the URIs used as the annotation values and insert them into the generated OWL-S *Groundings*. When an element used in an output message has no lifting schema mapping, Service Monitor will generate the XSLT scripts by analyzing the model references. However, for input messages, Service Monitor generates a single XSLT transformation script, and lowering schema mappings specified for the individual elements are ignored.

If the referent is an OWL class, XSLT derivation requires that all individuals of the class are known, and a mapping between string parameter values and the OWL individuals is specified through SAWSDL model references. An example of such a mapping can be found on lines 21-23 in Listing 5.9 in Section 5.6.

If the same XML schema type is used in several schema elements, it may be convenient to add the SAWSDL annotations to the type definition. Service Monitor will then interpret the annotations exactly similarly as if they had been attached to the schema elements using the type definitions. Thus, it is unnecessary to explicitly annotate the schema elements.

3.6 SWRL-based Semantic Web Service Composition

Section 3.5 proposed an approach to converting SAWSDL files referring to SWRL rules into OWL-S Processes. This section describes how the generated OWL-S descriptions can be used in semantic web service composition. The main difference to Section 3.3 is that now the condition and effect expressions may be formulated in SWRL instead of SPARQL, and therefore the planning solution presented in Section 3.3 is inapplicable. Furthermore, this section presents an enhanced planner

component attached to Service Monitor. The planner can solve even relatively complex problems without a separate restriction expression, which the planner presented in Section 3.3 typically requires for other than the most trivial planning problems.

While the proposed composition approach is primarily based on the assumption that the service pre- and postconditions are expressed using SWRL, it can support service descriptions using SPARQL as the expression language. However, the SPARQL expressions must be restricted to a relatively simple syntax that can be automatically translated to SWRL.

3.6.1 Composition Pattern Overview

The current version of Service Monitor supports the parallel achievement of several production goals. A new goal can be registered by invoking the *StartGoal* operation, which assigns the goal a unique identifier and initiates a process that first attempts to compose a solution plan for achieving the goal and then executes the plan. The process is automatically repeated until either the goal is achieved or the goal is terminated by invoking the *StopGoal* operation. The repeating of the process is required when either the AI planning algorithm fails to compose a solution plan for achieving the goal or the execution of the obtained plan fails.

Figure 3.11 illustrates the registration of a new goal and the subsequent goal achievement process through a sequence diagram. The main difference to the sequence diagram of Figure 3.3 is that now the registering of new goals and their achievement is asynchronous. Furthermore, Service Monitor sends *GoalStatusChanged* event notifications to registered subscribers as the status of a goal process changes. Table 3.7 summarizes the new operations in the Service Monitor interface.

3.6.2 Requirements

The requirements for using the enhanced planner are as follows:

- The semantic web service descriptions use SWRL or SPARQL (with some syntax restrictions) in expressing conditions and effects.
- The number of objects in the domain model is constant.

SWRL has a somewhat simple syntax. Hence, when OWL-S Process conditions and effects are expressed in SWRL, it is relatively effortless to convert the Processes into planning operators that include both negative and positive conditions, as well as negative and positive effects. Nonetheless, the requirement of using SWRL is not absolute, since it is possible to automatically translate other languages, such as SPARQL, into SWRL. On the other hand, some SPARQL constructs are somewhat

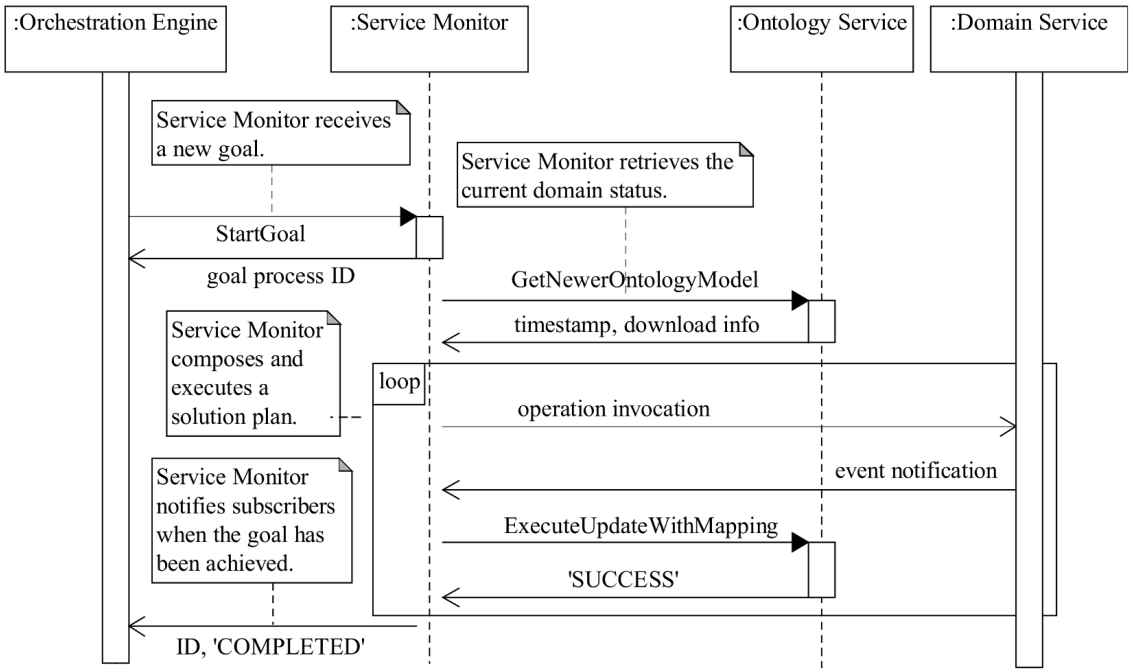


Figure 3.11: The *StartGoal* operation is invoked to initiate a new goal process.

Table 3.7: The Service Monitor interface provides operations for starting and terminating goal processes as well as for monitoring their progress.

Operation Name	**Inputs**	**Outputs**	**Purpose**
StartGoal	A goal (SPARQL ASK query)	Goal process ID (string)	Initiates a new goal process.
StopGoal	Goal process ID (string)	Response code ('SUCCESS' / 'FAILURE')	Terminates a goal process.
ListGoals	No input parameters.	List of goal process statuses: • ID (string) • Status ('RUNNING' / 'COMPLETED' etc.)	Retrieves the list of current goal processes.
GoalStatusChanged	No inputs (event notification)	Current process status: • ID (string) • Status ('RUNNING' / 'COMPLETED' etc.)	Notifies of process status changes.

challenging to translate into SWRL, and Service Monitor currently requires that such constructs are avoided if the SWRL-based planning approach is used.

The most critical requirement is that none of the OWL-S Processes creates new OWL instances in the domain model. In other words, the set of objects in the domain model must be numerable. The requirement allows the planning component to assign a unique ID number to each object. Thus, the planner can enumerate each statement built using the objects and predicates, as well.

3.6.3 Obtaining an AI Planning Problem

While the enhanced planner is conceptually based on PDDL, it internally uses a numeric representation of the domain and the planning operators. Nonetheless, the numeric model is used only in computations and can be rapidly converted into the human-readable PDDL syntax.

Before attempting to solve a planning problem, the planner instantiates all actions in the planning problem. The action instantiation includes first determining all of the applicable variable value combinations for each SWRL rule and then using each combination to create a new planning operator, whose conditions and effects refer only to ground instances. Thus, the number of the resulting planning operators is typically considerably larger than the original number of SWRL rules extracted from the OWL-S *processes*. After the instantiation, the effects and conditions of each operator are sets of statements, which are internally represented as numbers. The planning goals are instantiated similarly, and therefore a goal can be instantiated to a set of alternative goals, each based on a different variable value combination.

After the aforementioned instantiation procedure, the input data used by the planning algorithm includes the following:

- the set of all possible statements $S \subset \mathbb{Z}$
- the initial state $I \subset S$
- the set of goal state alternatives G, where each $g \in G$ is a tuple $\langle g^+, g^- \rangle$, so that
  - $g^+ \subset S$ is the set of positive goals
  - $g^- \subset S$ is the set of negative goals
- the instantiated actions A, where each $a \in A$ is a tuple $\langle pre^+, pre^-, post^+, post^- \rangle$, so that
  - $pre^+ \subset S$ is the set of positive preconditions
  - $pre^- \subset S$ is the set of negative preconditions
  - $post^+ \subset S$ is the set of positive postconditions

– $post^- \subset S$ is the set of negative postconditions

While different versions of the planner component can be developed, the forward search and backward search solutions appear the most viable. The backward search planner preprocesses the domain description by eliminating all negative preconditions from actions. The algorithm for eliminating negative preconditions is described in [24]. After the preprocessing step, the instantiated actions can be expressed as triples $\langle pre, post^+, post^- \rangle$, where $pre \subset S$ denotes the set of statements that must hold for the action to be applicable. However, the forward search algorithm described in the sequel includes no preprocessing phase.

3.6.4 The Domain-independent Planning Algorithm

Thus far, the forward search algorithm appears the most efficient solution, since it performs simple pruning of the state space by only considering those actions that can be relevant for the desired production goal. The state space pruning technique is essentially based on the concept of helpful actions described by Hoffmann and Nebel [29]. An action $a \in A$ is potentially relevant if it satisfies one of two conditions:

- The action directly achieves a proposition that is included in one of the alternative production goals: $\exists g \in G : post^+(a) \cap g^+ \neq \emptyset \vee post^-(a) \cap g^- \neq \emptyset$.
- The action directly achieves a proposition that is included in the conditions of another action: $\exists a_2 \in A : post^+(a) \cap pre^+(a_2) \neq \emptyset \vee post^-(a) \cap pre^-(a_2) \neq \emptyset$.

The set of potentially relevant actions remains constant for each goal, and hence it is sufficient to compute it only once after instantiating the actions.

The planning algorithm is based on state space search. Each state is identified by the set of statements that hold in that state. As each statement is enumerated, the states are essentially sets of integers. However, the representation can be compacted by only representing a state as a set of integers indicating the differences to the initial state.

While forward search can be conducted in both Bread-First Search (BFS) and Depth-First Search (DFS) manner, BFS is more applicable to the planning algorithm, as it guarantees that only minimal solutions are discovered.

Algorithm 2 contains the pseudo-code for the planning algorithm. The forward search planning algorithm starts from the initial state and considers all potentially relevant actions to compute the successor states. As all successor states are added to a First-In, First-Out (FIFO) stack of unprocessed states, the search space nodes are processed in BFS-manner. Once the successor states have all been computed, the algorithm retrieves the first state in the stack and continues search. The algorithm

Algorithm 2 The forward search algorithm prunes the state-space by only considering potentially relevant actions.

Require: $maxNodeCount \geq 0$
  $depth \Leftarrow 0$
  $Relevant \Leftarrow \{a \mid a \in A \wedge isRelevant(a)\}$
  $StartNode \Leftarrow I$
  $Graph.nodes \Leftarrow \{StartNode\}$
  $Unprocessed \Leftarrow \{StartNode\}$
  while $Unprocessed \neq \emptyset$ **do**
    choose the first node $Current \in Unprocessed$
    $Unprocessed \Leftarrow Unprocessed \setminus \{Current\}$
    for all $a \in Relevant$ **do**
      if $pre^+(a) \subseteq Current \wedge pre^-(a) \cap Current = \emptyset$ **then**
        $NewState \Leftarrow Current \cup post^+(a) \setminus post^-(a)$
        $Graph.nodes \Leftarrow Graph.nodes \cup \{NewState\}$
        $Unprocessed \Leftarrow Unprocessed \cup \{NewState\}$
        Create a link between $Current$ and $NewState$
        if $\exists g \in G \mid g^+ \subseteq NewState \wedge g^- \cap NewState = \emptyset$ **then**
          return $NewState$
        else if $|Graph.nodes| > maxNodeCount$ **then**
          return failure
        end if
      end if
    end for
  end while

terminates if the state satisfies the goal or if there are no more unprocessed states in the stack. The latter case indicates that there is no solution for the planning problem. However, typically the amount of computing resources available is limited. To avoid excessive use of resources, the search algorithm terminates after analyzing the number of nodes indicated by the *maxNodeCount* parameter, which may be set to, for example, 500000.

The output of Algorithm 2 is a state-space node that fulfills one of the alternative goals. The links in the search graph correspond to instantiated planning operators, each of which can be mapped to a source OWL-S Process and the appropriate selection of input parameter values. Hence, the planner can extract a sequence of semantic web service invocations that leads to a domain state fulfilling the goal expression. However, executing such sequential solution plans may be unnecessarily slow, since only one operation is executed at a time. Therefore, the planner component incorporates an additional solution plan refinement phase transforming the sequence into a tree structure. The tree has a single root node, which corresponds to the initial state and a single leaf node, which corresponds to the terminal node of a plan execution. The links between the nodes correspond to OWL-S processes

and the associated input variable bindings. Each non-root and non-leaf node must have one or more of input links and output links.

The plan execution begins from the root node of the plan tree. When all input actions of a node have been executed, the output actions are executed in parallel. The plan execution is completed when all input actions of the terminal node have been executed.

3.7 Cloud Resource Utilization Optimization

Utilization of resources leased from IaaS clouds can be elaborate. To reduce the workload, this section describes a web service that acts as a mediator between a computing cloud and the cloud users, thus facilitating cloud usage. Henceforth, the service will be called Cloud Gateway.

One instance of the Cloud Gateway service is started on each virtual machine leased from an IaaS cloud. The service instances then enable a user to effortlessly execute applications on the virtual machines. Moreover, the Cloud Gateway services can form networks spanning several virtual machines that may reside in separate computing clouds. Thus, when a Cloud Gateway is low on computing resources, it can delegate a request for starting a new application to another Cloud Gateway instance hosted by a less burdened virtual machine.

3.7.1 Adding and Executing Applications

Cloud Gateway provides operations for adding and removing applications to and from its application library as well as starting and terminating instances of the applications. Cloud Gateway assigns a unique string identifier to each application in its library and to each application instance started. The Cloud Gateway service interface includes operations that can be categorized into query operations, which take no input values, and effective operations, which cause Cloud Gateway to perform actions based on the input parameters. Table 3.8 lists the most important query operations, and Table 3.9 summarizes the effective operations.

Cloud Gateway has been particularly tested in scenarios where each application, when executed, creates and starts a web service compliant with the DPWS specification [17]. Thus, in the sequel, such applications are called server applications.

To facilitate the effortless transfer and execution of the server applications, they must be packaged into executable Java archive (JAR) files. Hence, Cloud Gateway can download the applications as single files. Furthermore, the applications can be executed on any platform that has a sufficiently new Java runtime environment installed.

An application can be executed by invoking the *StartApplication* operation and

Table 3.8: The Cloud Gateway service includes operations for querying the current service status.

Operation	Outputs
ListApplications	The list of applications available in the application library. The identifier, JAR file name and default arguments are listed for each application.
ListAll	The list of all application instances. The instance identifier, application identifier, command-line arguments and state (running or terminated) are listed for each instance.
GetResourceUsage	Numeric values indicating the amount of free memory, total memory, the number of CPUs, the system load average, as well as the current memory and CPU utilization thresholds.

passing the application identifier as an input. Optionally, a list of command-line arguments may be specified to override the default arguments. As a response, *StartApplication* returns the identifier assigned to the new application instance or 'FAILURE' if it fails to start the application.

Command-line arguments may contain certain keywords that Cloud Gateway expands before executing the corresponding application. Keywords are identified by enclosing them between '$#' and '#$'. For example, Cloud Gateway replaces each occurrence of the string '$#HOST#$' with the host machine network address.

A typical server application needs several seconds to deploy a set of web services. Web services compliant to the WS-Discovery specification [68] send Hello messages when they enter a network. Hence, Cloud Gateway listens to Hello messages originating from the host machine. Whenever Cloud Gateway receives such a message, it sends a *ServiceStarted* notification to all subscribed clients. The notifications allow the clients to determine server application start-up times.

Cloud Gateway allows multiple instances of each application to be executed in parallel. A running application can be terminated by executing the *TerminateApplication* operation. Since Cloud Gateway is able to terminate applications only in a forcible manner, the terminated applications must prepare for the abrupt termination of the underlying Java virtual machine and perform the necessary activities at such an event. For example, DPWS-compliant web services should broadcast WS-Discovery Bye messages when leaving the network. The Bye messages allow clients to automatically detect when the web services become unavailable.

3.7.2 Resource Consumption

The number of applications that a Cloud Gateway is able to start depends on the amount of hardware resources available to the virtual machine hosting the service. Therefore, Cloud Gateway must use some metrics for determining the amount of idle

Table 3.9: The Cloud Gateway service includes operations for managing the set of available applications as well as executing and terminating them.

Operation	**Inputs**	**Outputs**
AddApplication	**location** - the URL from which the JAR file can be read **parameters** - the default command-line arguments	The identifier assigned to the application or 'FAILURE' if reading the JAR file from the specified URL fails.
RemoveApplication	**id** - the application identifier	'SUCCESS' or 'FAILURE' if no application with the specified identifier exists, or if a running instance of the application exists.
StartApplicationInNet	**id** - the identifier of the application to start **parameters** - the list of command-line arguments, if empty, the default arguments will be used	The identifier assigned to the new application instance or 'FAILURE' if starting the application failed. The endpoint URI of the Cloud Gateway that started the application.
TerminateApplication	**id** - the identifier of the application instance to terminate	'SUCCESS' if the application was running, otherwise 'FAILURE'.
SetThresholdInNet	**MemoryThreshold** - a floating point value between 0 and 1 **CPUThreshold** - a non-negative floating-point value	'FAILURE' if the threshold values are outside the allowed ranges, otherwise 'SUCCESS'.
RegisterCloudGateway	**URI** - the endpoint URI of the Cloud Gateway instance to register	'FAILURE' if the Cloud Gateway service had already been registered, otherwise 'SUCCESS'.
DeregisterCloudGateway	**URI** - the endpoint URI of the Cloud Gateway instance to deregister	'SUCCESS' if the Cloud Gateway service had been registered, otherwise 'FAILURE'.

resources. Furthermore, it must compare the resource utilization levels measured to threshold values indicating the maximum levels allowed. Cloud Gateway accepts a request to start an application only if the current resource utilization levels are below those indicated by the threshold values.

The metrics that most clearly define the resource utilization of a virtual machine

are the random access memory (RAM) and central processing unit (CPU) usage. In Linux systems, the percentage of RAM used can be measured fairly effortlessly by examining the contents of the virtual */proc* file system. The CPU usage level is more problematic to determine but can be derived from the system load average, which can also be determined from the */proc* file system. The load average represents the number of processes that are either in execution or queuing for CPU time. Hence, the higher the value, the more burdened the CPU is. If the load average is equal to the number of CPUs, CPU utilization is optimal [115]. Hence, to calculate a value for the CPU utilization level, Cloud Gateway divides the system load average with the number of CPUs.

If either the RAM or CPU usage value calculated is higher than the corresponding threshold value, Cloud Gateway rejects all requests to start new application instances. The threshold values can be specified by invoking the *SetThreshold* operation.

3.7.3 Cloud Gateway Networks

The system resources of a virtual machine will inevitably be exhausted if several application instances are executed on the machine. Therefore, Cloud Gateways residing on separate machines can form networks to balance the load between several machines. For this purpose, the Cloud Gateway service interface includes the operations *RegisterCloudGateway* and *DeregisterCloudGateway*, which allow the registering and deregistering of partner Cloud Gateways that will be used in workload balancing. The *StartApplicationInNet* operation will execute the application locally on the host machine only if the resource utilization is within allowed boundaries. Otherwise, the *StartApplicationInNet* operation is recursively invoked on the partner Cloud Gateways to find one that is able to service the request. Similarly, the *SetThresholdInNet* is a recursive version of the *SetThreshold* operation.

The sequence diagram in Figure 3.12 presents a typical use scenario of Cloud Gateway. The Client object in Figure 3.12 can be an autonomous software agent or a software tool operated by an end user. In the beginning of the example sequence, the client registers Cloud Gateway 2 to Cloud Gateway 1 to form a Cloud Gateway network. Then, the client registers a new server application to Cloud Gateway 1. Once Cloud Gateway 1 has downloaded the application, the client executes it in the cloud by invoking the *StartApplicationInNet* operation. Because Cloud Gateway 1 is low on computing resources, it delegates the request to Cloud Gateway 2, which then executes the application, effectively deploying a new web service. The response to the original *StartApplicationInNet* request includes the endpoint URI of the selected Cloud Gateway instance. Finally, the client requests Cloud Gateway 2 to terminate the server application to release the computing resources for future use.

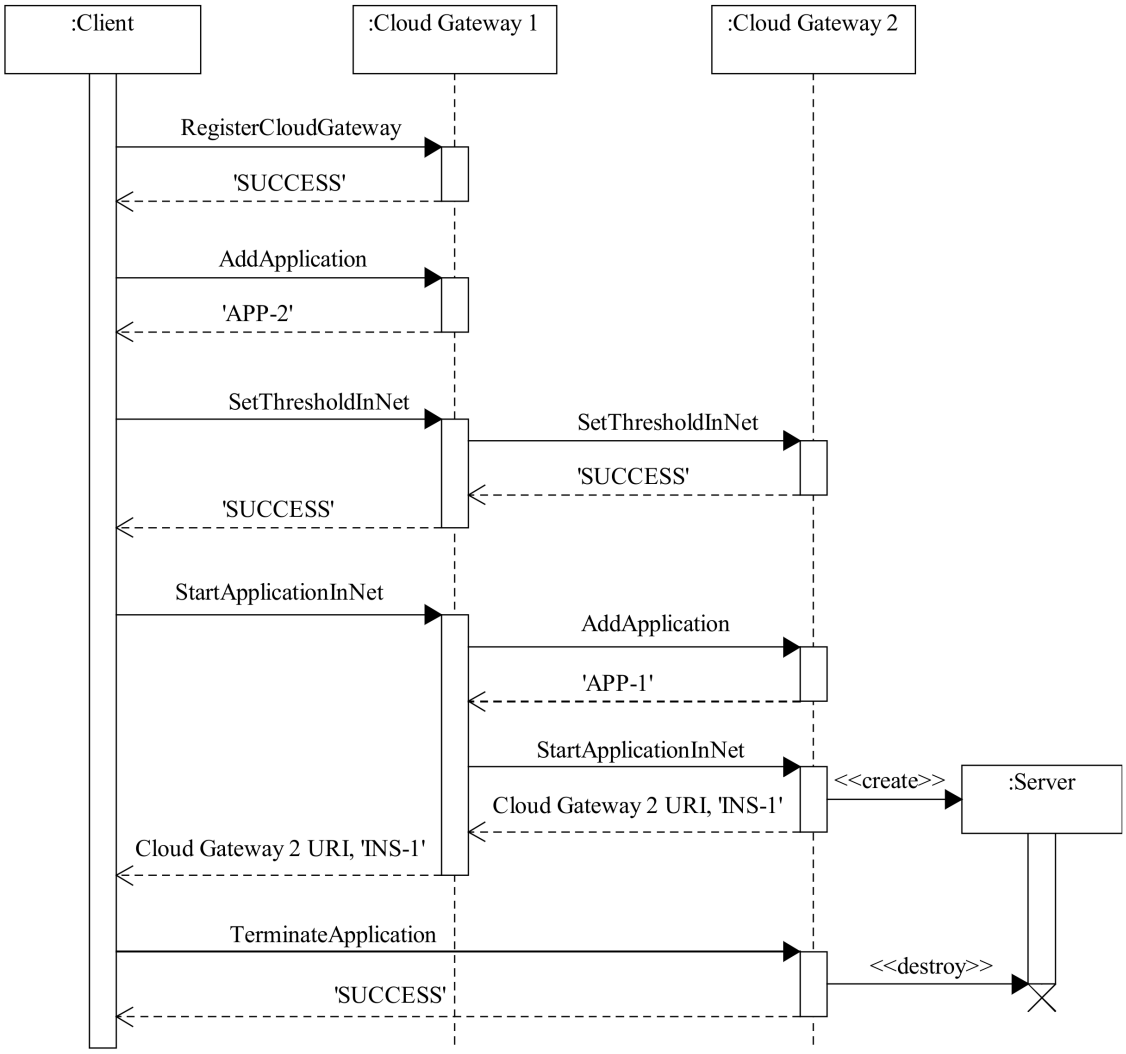


Figure 3.12: A typical use scenario of Cloud Gateway includes starting a web service and terminating it after use to conserve resources.

If a Cloud Gateway selects another service instance in the network to execute an application, it must first ensure that the other instance possesses a copy of the application and obtain the application identifier by invoking the *AddApplication* operation. In the sequence diagram of Figure 3.12, the other instance returns the application identifier 'APP-1' as part of its response to the service request.

3.8 Summary and Conclusions

Table 3.10 summarizes the methods presented by each of the sections in this chapter. While both Section 3.3 and 3.6 address automated service invocation, the latter section presents a somewhat more advanced approach, in which the service condition and effect expressions are processed to allow more efficient planning. However, the latter section also sets some additional requirements on the semantic service descriptions. In addition, Table 3.10 lists the framework web services used in each

Table 3.10: The methods presented in this chapter address different problems.

Section	**Problem Addressed**	**Framework Web Services**	**Application Section**
3.1	Manual service workflow prescription	• Orchestration Engine	5.2
3.2	Automated service discovery	• Ontology Service • Ontology Manager • Orchestration Engine • Service Monitor	5.3
3.3	Automated service invocation	• Ontology Service • Orchestration Engine • Service Monitor	5.4
3.4	The dynamic updating of a world view	• Ontology Service • Ontology Manager / Service Monitor	5.5
3.5	Automated semantic web service description development	• Service Monitor	5.6
3.6	Automated service invocation, advanced issues	• Ontology Service • Orchestration Engine • Service Monitor	5.7
3.7	Computing Cloud utilization, web service deployment	• Cloud Gateway	5.8

of the methods and specifies the section of Chapter 5 in which each method is applied.

This chapter has proposed a framework of software tools, Orchestration Tools, and Figure 3.13 categorizes the individual software tools into two groups: web services and applications that only include a graphical user interface (GUI). Chapter 4 describes each of the tools with particular focus on its implementation.

The arrows in Figure 3.13 indicate the dependencies between the tools. While some of the tools substantially depend on others, two user-interface applications and the Cloud Gateway web service have no dependencies on other tools.

SWRL Planner is a GUI application intended for visual experimenting on different implementations of the planner component discussed in Section 3.6. Provided that the planners implement a common Java interface, they can be effortlessly plugged into SWRL Planner for testing. While the planner components are typically intended for Service Monitor, SWRL Planner includes no direct dependencies on other tools, and vice versa.

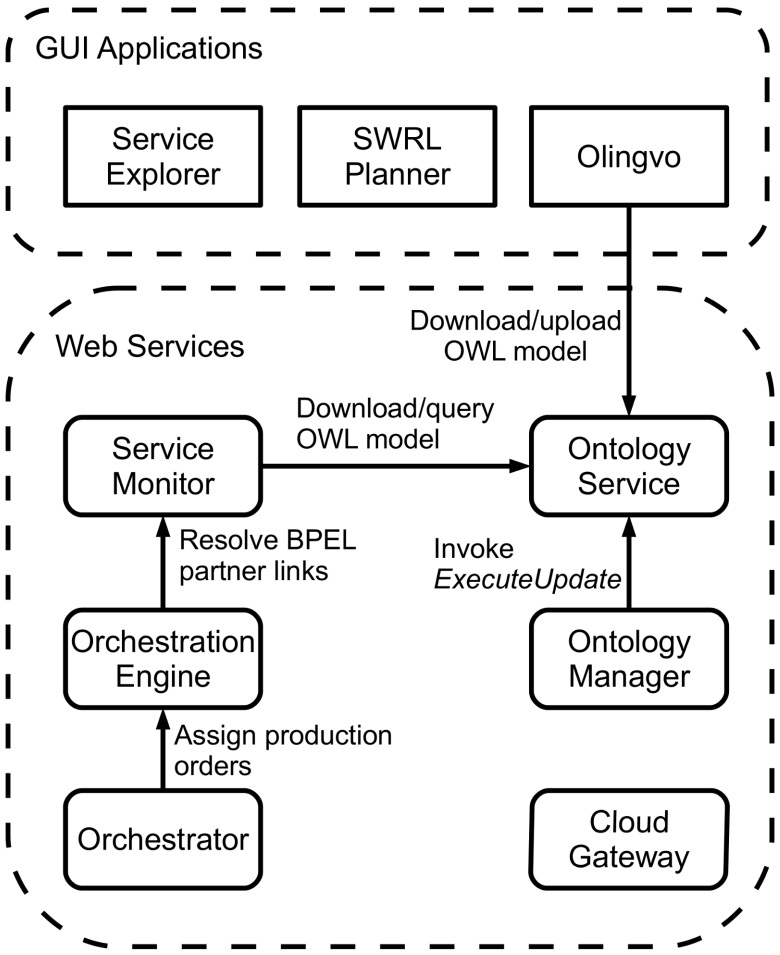


Figure 3.13: The proposed Orchestration Tools framework is composed of a set of collaborating web services and applications.

Ontology Service is an integral data container used by two other web services and the Olingvo GUI application. Olingvo is an OWL model engineering tool which can download a copy of the current OWL model from Ontology Service as well as upload the OWL model currently open in Olingo to Ontology Service. Ontology Manager is intended to update the OWL model hosted by Ontology Service based on update rules specified by the user. Service Monitor downloads the current OWL model from Ontology Service before attempting to solve a planning problem, in which the model represents the initial state. Furthermore, Service Monitor updates the OWL model hosted on Ontology Service based on update rules automatically extracted from semantic web service descriptions. In addition, Service Monitor monitors the plan execution progress by sending queries to Ontology Service.

The Orchestrator web service tasks Orchestration Engine service instances with executing BPEL processes included in production orders. The Orchestration Engine service depends on the Service Monitor web service to determine the semantic web services that match BPEL partner links.

4. IMPLEMENTATION

This chapter describes the implementation of the software tools that aim to solve the issues identified in Section 1.2. Together, the tools form a set called Orchestration Tools. The tools have been developed based on object oriented programming (OOP). In OOP, software components consist of classes, which are instantiated at run-time. In the development of Orchestration Tools, Java has been used as the sole programming language due to its platform-independence and special emphasis on OOP. The Java Swing library provides the foundation for all of the graphical user interfaces in the software tools.

4.1 Service Explorer

Service Explorer is a graphical user interface application that makes it possible to discover available web services and invoke their operations. Figure 4.1 depicts the typical use cases of Service Explorer.

Before communicating with web services, Service Explorer must discover them through the WS-Discovery protocol. While the Orchestration Tools API includes a Java interface facilitating the development of custom WS-Discovery protocol implementations, the user can configure Service Explorer to use third-party WS-Discovery implementations. Although Service Explorer defaults to a built-in implementation, new discovery implementations can be dynamically added through the plug-in mechanism.

Once the user has initiated service discovery, the discovered services appear in a tree view. The user may then request Service Explorer to process and display the service WSDL descriptions.

Service Explorer is able to invoke operations and receive event notifications only from web services whose WSDL descriptions it has successfully processed. WSDL processing is initiated automatically whenever the user selects a web service in the tree view. The user may then view the available WSDL port types and operations in another tree view.

When the user chooses to invoke an operation, Service Explorer first displays a dialog where the user must specify the input parameters for the operation. Finally, Service Explorer displays the content of the potential response message received from the service.

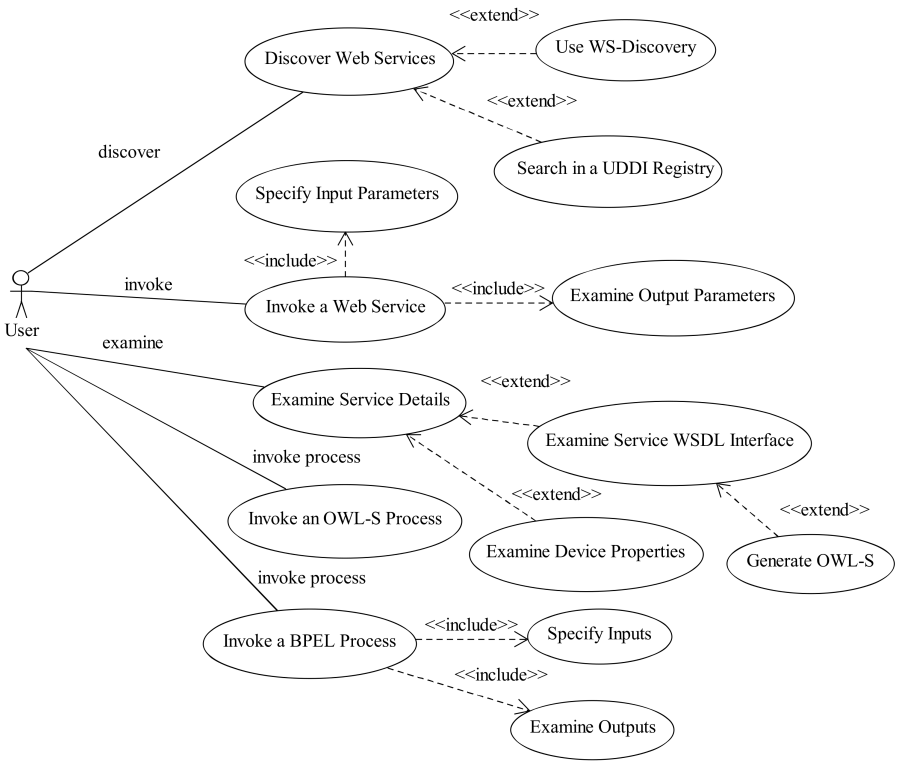


Figure 4.1: Service Explorer allows a user to invoke web services.

Service Explorer includes additional utilities for various purposes, such as executing BPEL processes, testing XSLT scripts to transform XML fragments, and generating OWL-S processes from web service operations.

While Service Explorer allows OWL-S processes to be executed, the current support for composite OWL-S processes is limited.

The use of Service Explorer in executing BPEL processes and invoking web services is discussed in Section 5.2.2 as well as in [81].

4.2 Olingvo

Olingvo is a graphical user interface application allowing the creation, browsing, and editing of OWL ontologies. Figure 4.2 depicts the typical use cases of Olingvo.

The OWL ontologies are stored in the RDF/XML format to OWL files. To specify the file to load, the user may either use a file navigator dialog or type the URL of the file.

The development of Olingvo has been inspired by the Protégé[1] ontology editor, and the two applications share several features. The main motivation for developing Olingvo despite the pre-existing OWL editors is the ease of fixing problems and

[1] Available at http://protege.stanford.edu/. Accessed on 2014-06-18.

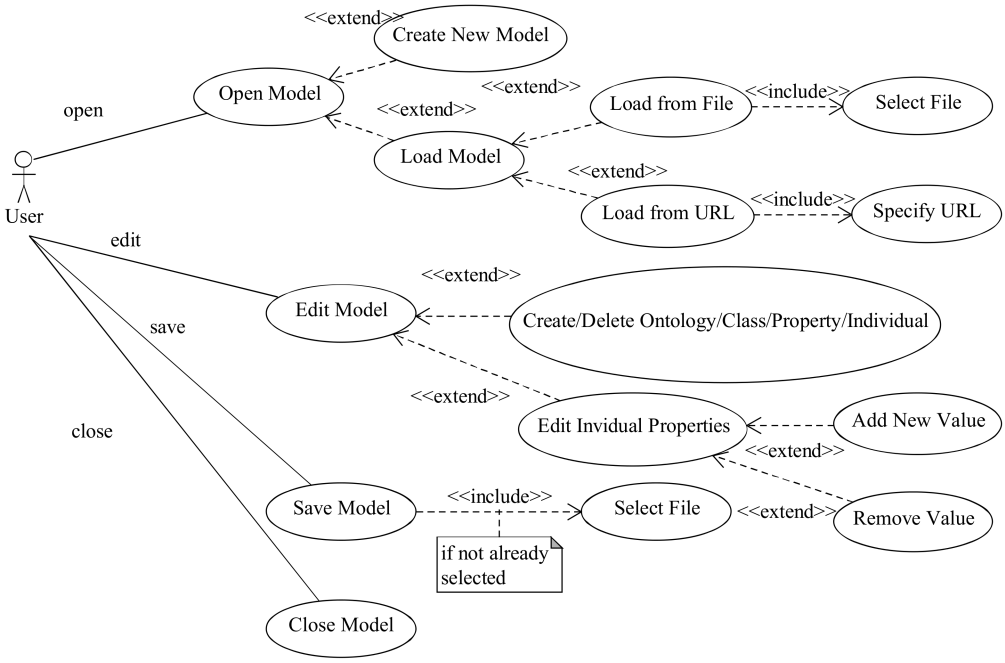


Figure 4.2: Olingvo provides a user interface for creating, browsing, and editing OWL models.

adding new features to in-house developed software. For example, Olingvo has support for evaluating SPARQL queries and executing SPARQL/Update expressions. Furthermore, Olingvo can interact with the Ontology Service, which is one of the web services in the Orchestration Tools framework.

Olingvo extensively relies on the Apache Jena APIs[2]. Indeed, Olingvo can be regarded as a GUI layer over the Jena APIs.

4.3 Ontology Service

Ontology Service is a web service that hosts an OWL model. Typically, the model is composed of a static TBox describing the concepts in an application domain and a dynamic ABox describing assertions. Hence, Ontology Service stores a view of the current world state. In addition, the web service interface provides read and write access to the OWL model, which makes it possible to update the world view as changes occur and to make decisions based on the current world state. While Ontology Service provides no graphical user interface for direct user access, the service is extensively used by other software-based actors in the Orchestration Tools Framework.

Ontology Service supports all SPARQL query types, except for the DESCRIBE type. However, the service interface provides different operations for invoking each

[2] http://jena.apache.org/

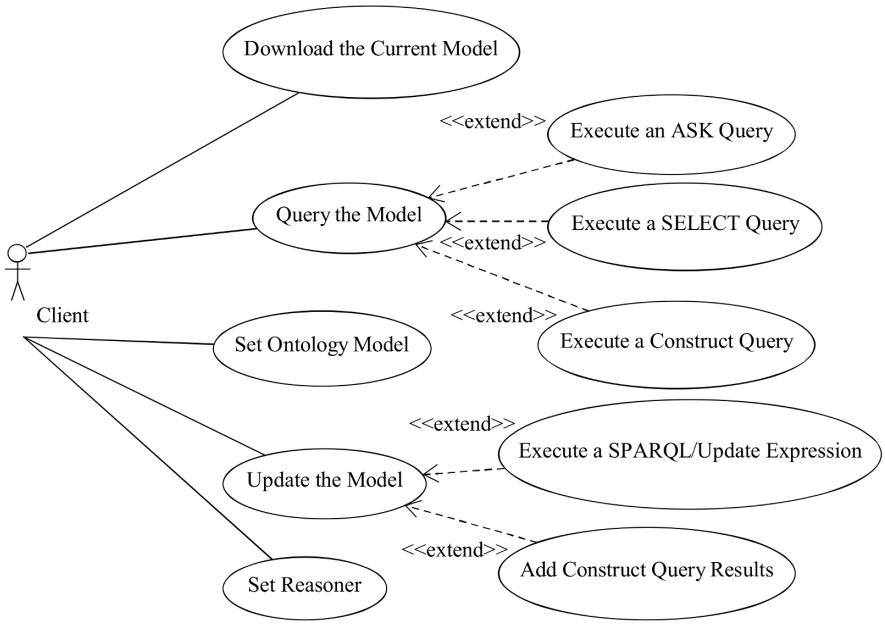


Figure 4.3: Ontology Service provides a web service interface through which other software actors may query and update the hosted ontology model.

type of query.

Figure 4.3 illustrates the use cases for Ontology Service. In the figure, the client actor is a web service client, which may be any software actor. The client may also be a human user communicating with Ontology Service through a web service client application. However, the Ontology Service interface is primarily designed for use by automated actors.

Sections 3.2 and 3.4.1 describe Ontology Service in detail. In addition, Sections 5.3, 5.4, 5.5, and 5.7 present application examples involving Ontology Service.

4.4 Orchestration Engine

Orchestration Engine is a web service that executes BPEL processes. As its name implies, Orchestration Engine is one of the core components of the Orchestration Tools Framework. However, with the adoption of semantic web service descriptions, which allow Service Monitor to automatically compose and invoke web services to achieve goals, Orchestration Engine has become a somewhat optional component.

Nonetheless, Orchestration Engine remains invaluable in invoking manually composed service workflows. Indeed, such execution flows are necessary when the domain web services lack appropriate semantic descriptions, which typically renders the services inapplicable to automatic service composition.

Similarly to Ontology Service, Orchestration Engine has been developed primarily for use by other software-based actors, such as the Orchestrator service presented

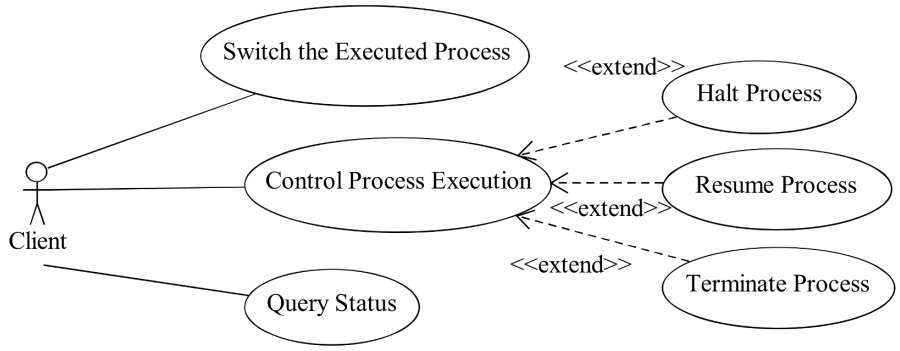


Figure 4.4: Orchestration Engine provides a web service interface for executing BPEL processes.

in Section 4.5. Therefore, Orchestration Engine lacks a graphical user interface and only displays minimal status windows without user control components. Instead, the functionality of Orchestration Engine can only be accessed through the web service interface. Figure 4.4 illustrates the typical use cases accessible through the interface.

The client actor in Figure 4.4 can be any software component sending request messages to the web service. For example, the executed BPEL process can be switched using a dedicated Orchestration Engine Client application, which includes a graphical user interface. Furthermore, Section 4.5 discusses a web service that acts as a client for Orchestration Engine.

4.5 Orchestrator

Orchestration Engine is unable to autonomously compose service execution workflows and requires that another software component supplies the executed BPEL processes. The Orchestrator service acts as a mediator between the workflow prescription source, such as an Enterprise Resource Planning (ERP) system, and a set of Orchestration Engine services. When it is first deployed, Orchestrator scans for an ERP service, and, if it discovers such a service, subscribes to be notified of new production orders.

Upon receiving a new production order, Orchestrator tasks an idle Orchestration Engine with servicing the order. In addition, Orchestrator provides both a web service interface and a graphical user interface for monitoring and controlling the execution status of orders. Figure 4.5 illustrates the use cases of the Orchestrator service. The ERP service in the diagram may be substituted by a human operator using the Orchestrator GUI.

The Orchestrator web service interface provides no request-response type operations for querying the status of production orders. Instead, the service sends event

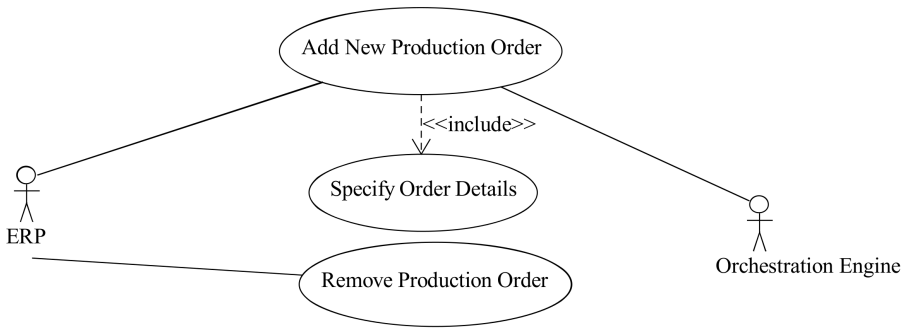


Figure 4.5: Orchestrator tasks Orchestration Engines with servicing production orders sent by an ERP service.

notifications through the WS-Eventing protocol. A notification is sent to each registered subscriber whenever the execution status of a production order changes.

4.6 Service Monitor

Service Monitor is the member of the Orchestration Tools framework services responsible for the composition of semantic web services to achieve goals formulated as SPARQL ASK queries. While the service is mainly intended to interact with other software actors through web service interfaces, it additionally includes a GUI for direct user interaction. Figure 4.6 depicts the main use cases accessible through both the WSDL service interface and the GUI.

Through the GUI, or the web service interface, the user may submit new goals for Service Monitor to achieve. The GUI shows all submitted goals in a table. Selecting a goal in the table will display the potential solution plan and its execution status in a graph view. The user may then freely pause and resume the plan execution. Stopping a plan execution will move the goal process into the 'TERMINATED' state, and Service Monitor will stop pursuing the goal.

To achieve goals, Service Monitor must be aware of semantic web services. Therefore, Service Monitor automatically scans the network during the initialization phase. In addition, Service Monitor constantly listens for Hello and Bye messages from new services entering the network and discovered services leaving the network. Nonetheless, the user may at any time re-initiate WS-Discovery through the main menu bar in the Service Monitor GUI.

In addition to fulfilling goals, Service Monitor reacts to event notifications from domain web services and updates the domain model at Ontology Service accordingly. However, Service Monitor automatically extracts the update rules from semantic web service descriptions, and the GUI includes no user controls for the processing of event notifications.

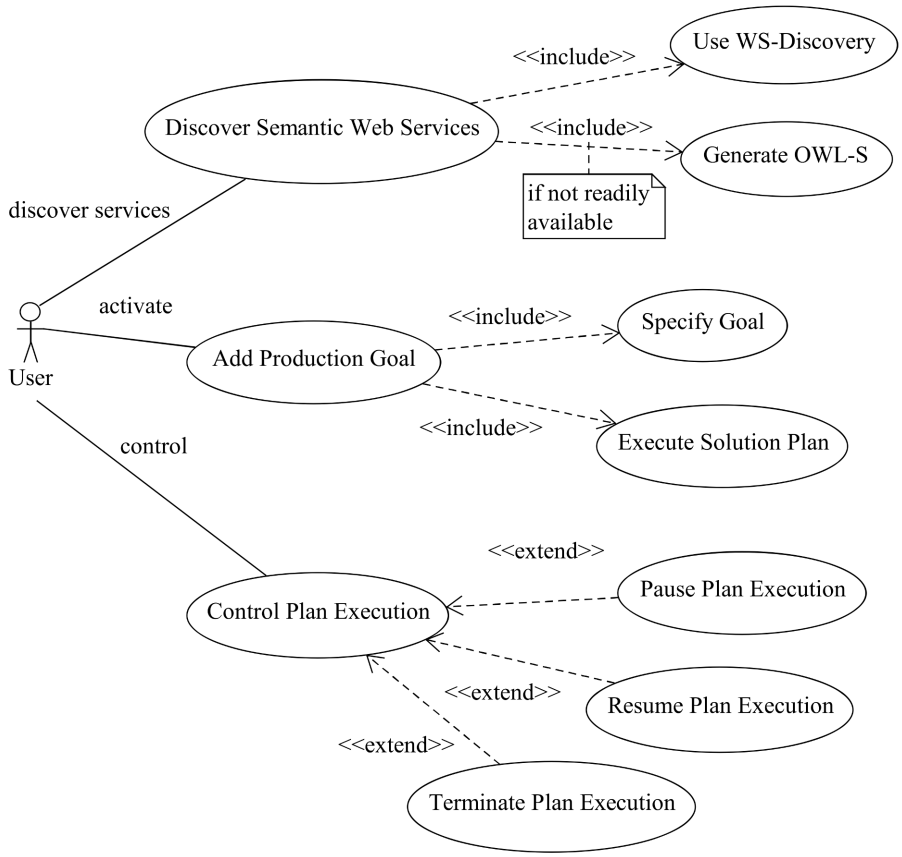


Figure 4.6: Service Monitor is a web service that additionally provides a graphical user interface.

4.7 SWRL Planner

Service Monitor can use various AI planner implementations in composing solution plans. As different planners may fail to solve some planning problems, it is invaluable to be able to observe planner performances. SWRL Planner provides a graphical user interface, through which a developer may experiment with different planning problems and examine the state space and solution plan generated by each planner.

The input data for SWRL Planner includes a goal, which is entered as a SPARQL ASK query in one of the GUI text areas, a domain model, which is loaded from an OWL file, and planning operators, which can be extracted from either the SWRL rules defined in an OWL model or OWL-S Processes. After solving a planning problem, SWRL Planner outputs two graphs representing the visited state space and the obtained solution plan. Figure 4.7 illustrates the use cases of SWRL Planner.

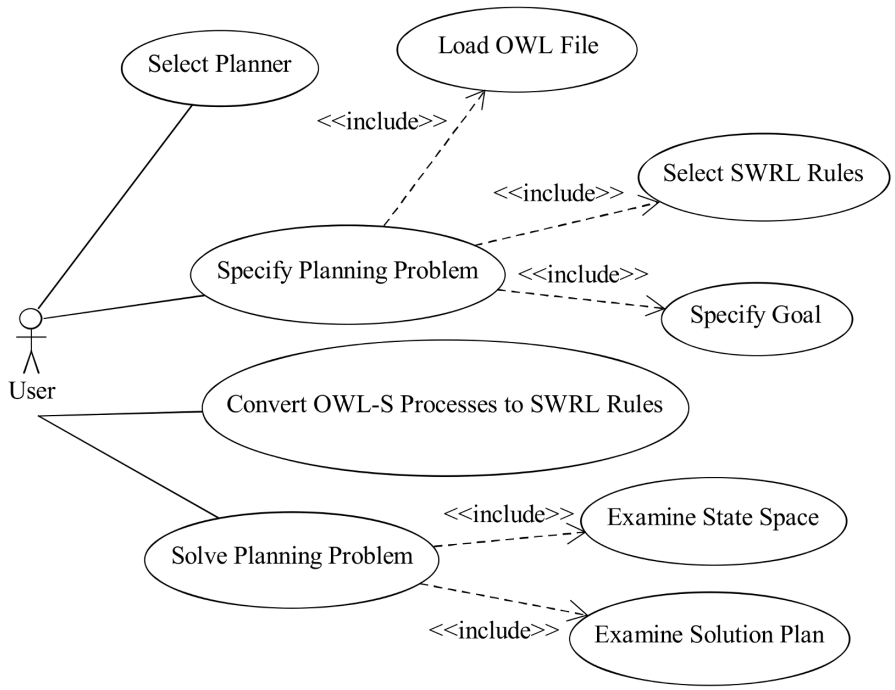


Figure 4.7: SWRL Planner provides a graphical user interface for testing different planner implementations.

4.8 Ontology Manager

Ontology Manager reacts to event notifications from web services and updates the domain model hosted by Ontology Service. Unlike the Service Monitor discussed in Section 4.6, Ontology Manager includes no capabilities to achieve goals. The web service performs the domain model updating based on update rules specified by the user. Hence, the approach is fully applicable without semantic web service descriptions.

Initially, Ontology Manager was a client application for Ontology Service. However, to allow other software agents to communicate with Ontology Manager, it was later transformed into a web service. Ontology Manager is typically connected to exactly one Ontology Service instance, to which Ontology Manager sends update requests based on the user-specified update rules.

The user may create and edit update rules through the Ontology Manager GUI controls. In addition, rules can be saved and loaded from an XML file. Figure 4.8 shows the main use cases accessible through the GUI.

Ontology Manager is typically in either the *passive* or *active* mode. In the former mode, the user may create, edit, and delete update rules, but all rules are disabled. In the latter mode, the update rules are uneditable, but Ontology Manager will apply the rules in updating the domain model hosted by Ontology Service.

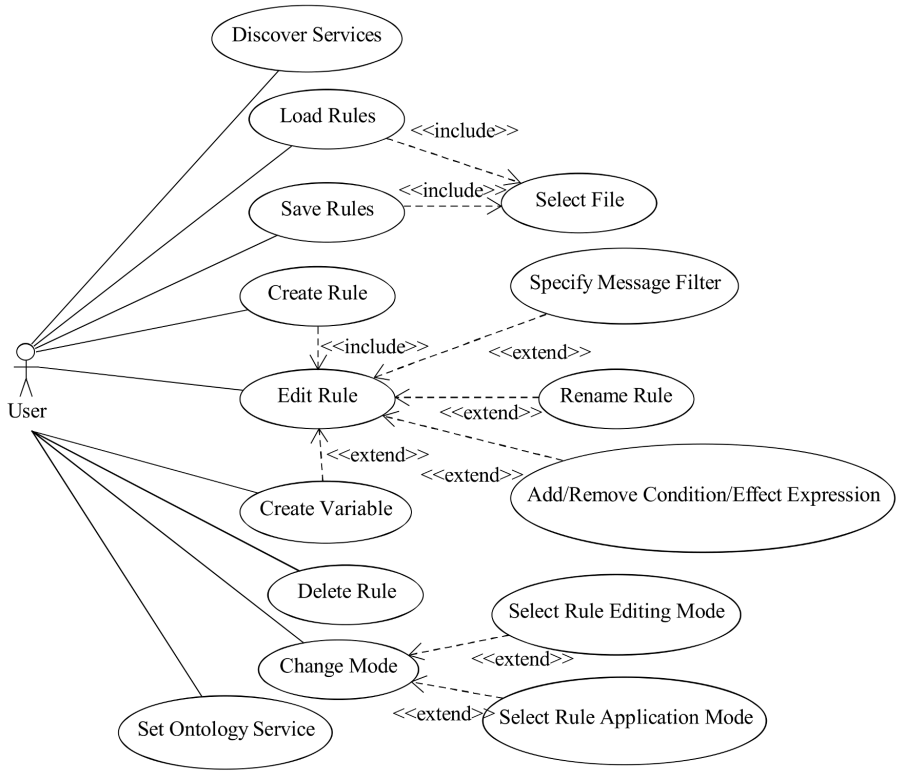


Figure 4.8: The Ontology Manager GUI provides fine-grained access to domain model update rules.

The more fine-grained rule-editing operations, such as creating a new update rule, are only available through the GUI. Nevertheless, the Ontology Manager web service interface provides access to all other operations, such as the loading of a new set of update rules from a URL as well as switching between the active and passive modes.

Ontology Manager is described in Section 3.4.2. In addition, dynamic domain model updating is discussed in [83].

4.9 Cloud Gateway

Cloud gateway optimizes the use of computing cloud resources by assessing the workload of the virtual machines available. As such, Cloud Gateway is a web service including no graphical user interface, except for a simple status window. Nonetheless, the Cloud Gateway client application, which is called Cloud Gateway Performance Test, provides a simple GUI.

The Cloud Gateway service is primarily intended to be deployed as part of a larger group in which each service instance is running on a separate virtual machine leased from a computing cloud. Together, the group forms a network of Cloud Gateways. Figure 4.9 illustrates the use cases of the Cloud Gateway service.

The Cloud Gateway service interface allows new applications to be registered

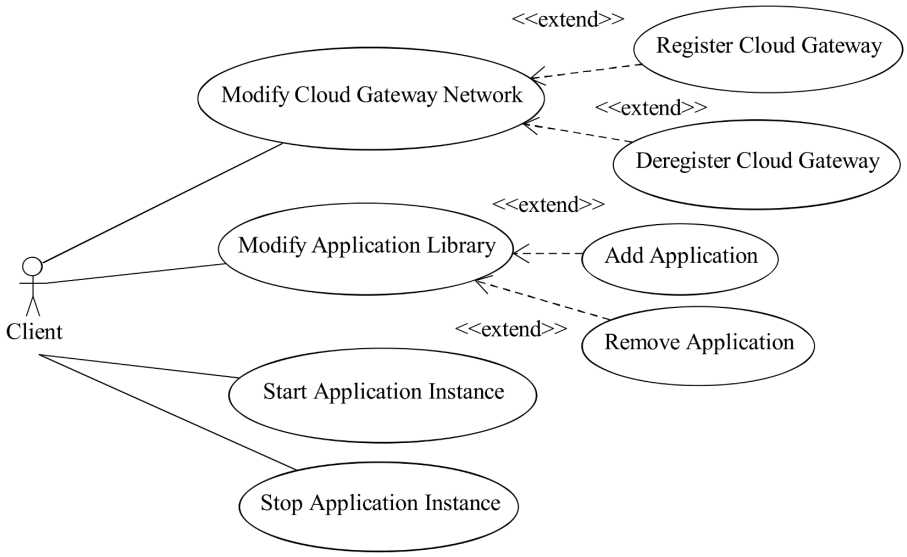


Figure 4.9: The Cloud Gateway service interface makes it possible to extend the Cloud Gateway network and deploy applications on the controlled cloud resources.

to the application library and instances of the applications already registered to be started. When a new application instance is started, the Cloud Gateway group collaboratively selects the virtual machine with the highest amount of idle resources, thus optimizing resource utilization.

While the Cloud Gateway approach is described in Section 3.7, Cloud Gateway and the client application were originally presented in [82].

4.10 Implementation APIs

The Orchestration Tools are implemented in Java, for which there exists a multitude of libraries facilitating tasks that would be laborious to program. As many of these libraries are under permissive licenses, they have reduced the programming effort of Orchestration Tools.

Several of the Orchestration Tools, particularly Ontology Service and Olingvo, frequently process OWL models. While developing such infrastructure code would be laborious, open-source OWL Java libraries are readily available. While there are several Java OWL APIs available, the Orchestration Tools only use the Apache Jena framework APIs[3]. In particular, all OWL-related activities, such as executing SPARQL queries, as well as reading and writing OWL data, are performed through the Jena RDF and ARQ APIs due to their extensive support for various OWL features as well as SPARQL and SPARQL/Update expression facilities.

Pellet[4] is an OWL reasoner that is compatible with the Jena RDF API. Therefore,

[3] Available at http://jena.apache.org/. Accessed on 2013-11-12.
[4] Available at http://clarkparsia.com/pellet/. Accessed on 2013-11-12.

Ontology Service and Olingvo include a dependency to Pellet. Although the use of Pellet is currently disabled by default in both of the tools, the reasoner can be activated through the Ontology Service WSDL interface and the Olingvo graphical user interface.

A web service implementation requires a framework that enables the service to, for example, be discovered through the WS-Discovery protocol and respond to WSDL operation invocations. All web services described in this dissertation have initially been implemented on one or several DPWS toolkits for Java[5,6]. However, the Orchestration Tools source code presently includes a web service framework based on core Java components. The new web service framework supports the DPWS specification, and the Orchestration Tools services have fully been retrofitted to rely on the new framework instead of third-party web service stacks.

Several web services in the Orchestration Tools framework host web resources. For example, each of the services hosts a WSDL description document, and Cloud Gateway service can additionally deploy web services packaged in WAR (Web ARchive) files. To be able to publish the resources, the services rely on the Jetty Web Server[7]. The services run Jetty in embedded mode, which means that the services launch one or several Jetty web servers at start-up and stop them when the services are shut down.

[5] DPWS4J Toolkit. Available at https://forge.soa4d.org/projects/dpws4j/. Accessed on 2013-12-18.
[6] WS4D.org Java Multi Edition DPWS Stack. Available at http://ws4d.e-technik.uni-rostock.de/jmeds/. Accessed on 2013-12-18.
[7] Available at http://www.eclipse.org/jetty/. Accessed on 2014-01-14.

5. APPLICATION EXPERIMENTS

This chapter will present application examples of the methods proposed in Chapter 3.

5.1 Application Domains

This section will briefly describe the domains of the application examples presented in the subsequent sections.

5.1.1 The Light Tower Monitoring Device

The light tower service encapsulates a simple signal tower device occasionally included in production machinery to report the status of the equipment. Figure 5.1(a) shows a photo of such a device. This scenario has been tested using both a purely virtual web service and an actual light tower device controlled by an Inico S1000[1] controller. Figure 5.1(b) shows an S1000 controller installed to an actual production system. The controller in the figure is connected to a network via a black Ethernet cable. In addition, the controller is connected to a power source as well as signal inputs and outputs.

When the controller is used with the signal tower device, the digital outputs of the controller are connected to the inputs of the signal tower, so that the controller can activate and deactivate the signal tower segments. The controller has been programmed to host a web service, whose interface includes simple operations, such as *SwitchRed* and *GetRedStatus*, for activating and deactivating the light segments as well as for querying the current activation states. In addition, the interface contains notification operations, such as *RedStateChanged*, which the service uses for sending event notifications when changes occur in the light states. The controller is connected to the same local network with the PC hosting the Ontology Service and Ontology Manager, which enables the other web services to effortlessly discover the light tower service.

The S1000 controller complies with the DPWS specification. Hence, software tools can interact with the web services implemented on the controller similarly to PC-based services.

[1]Inico Technologies. Available at `http://www.inicotech.com/`. Accessed on 2013-5-15.

(a) The light tower web service represents a type of signal tower device used in production systems.

(b) A production device can be viewed as a web service when it is connected to an appropriately programmed controller.

Figure 5.1: The light tower scenario involves a light tower device controlled by an RTU.

5.1.2 Conveyor Device Control

When six unidirectional conveyors are connected in a loop as depicted in Figure 5.2, a pallet may traverse the loop in only one direction, which is indicated by the arrows in the figure.

The production equipment in the example system comprise six conveyor segments of the same type. Hence, each of the devices exposes a similar web service interface. The conveyor devices in the production line are controlled by STBs provided by Schneider Electric, and the domain is in fact a fragment of the production line which will be presented in Section 5.1.3.

5.1.3 The Socrades Production Line

The approach proposed in Section 3.3 has been tested on a virtual model of a production line from the SOCRADES project [95]. While the application experiments discussed in section 5.4 have only been performed on the model, a more detailed description of the original production line can be found in [112]. The virtual system consists of web services implemented using a DPWS toolkit for Java. While the web services in the original system are hosted on the actual controller devices, the virtual services are hosted on a PC. The production line includes 29 conveyor segments, three of which are lifters, as well as five workstations. As each of the devices is abstracted as a web service hosted by the controller devices, the system includes 34 web services in total. Figure 5.3(a) shows a schematic of the demonstration line, and Figure 5.3(b) shows a top-down map of the individual conveyor segments. The two-dimensional map highlights the workstation locations with dotted rectangles and the allowed pallet movement directions with arrows.

The five conveyor segments in the lower part of Figure 5.3(b) represent the lower

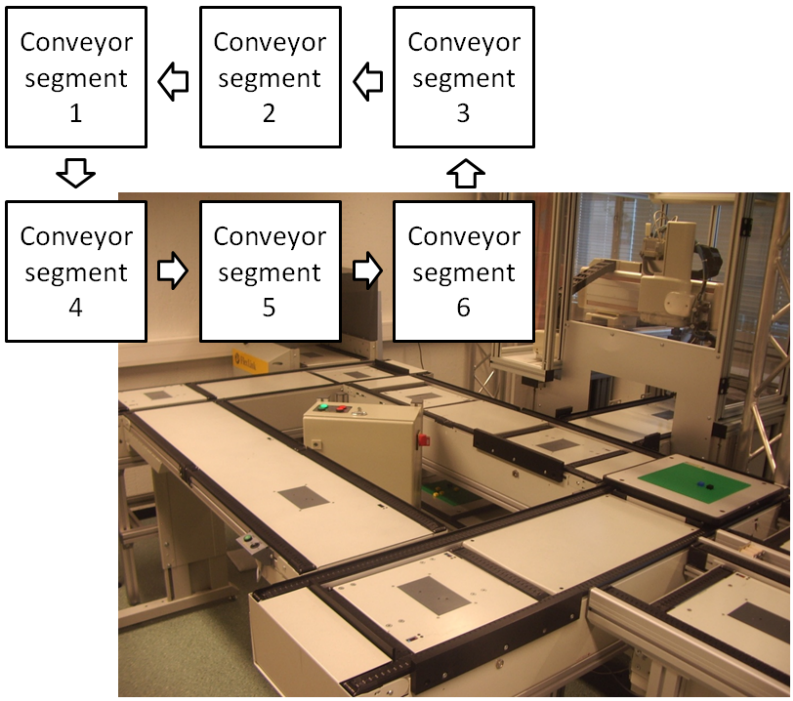


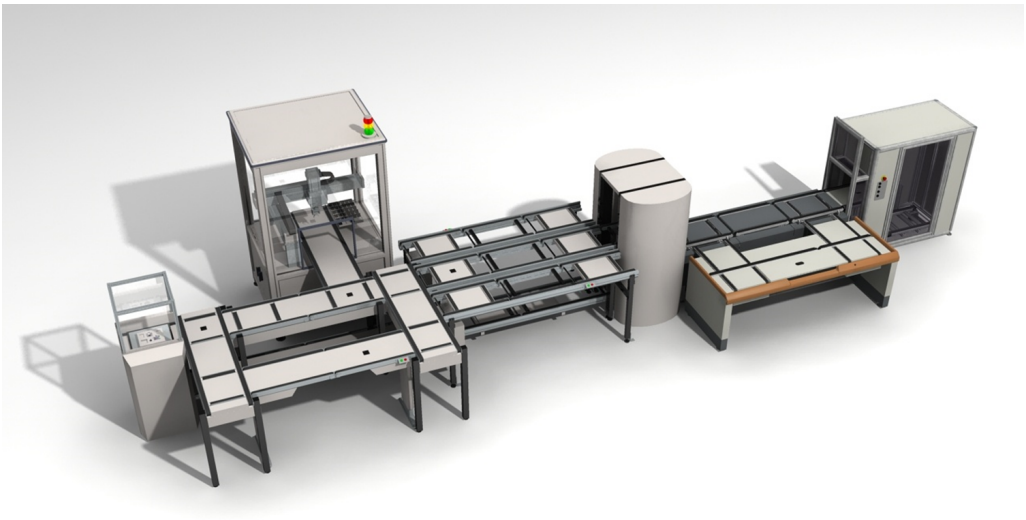
Figure 5.2: The six web services represent a loop of conveyor devices in the actual production line.

conveyor frame, which allows a pallet to be transported to an earlier position on the line. The start lifter, `S_ML_L1`, allows a pallet to be transported from the lower conveyor frame to the upper conveyor frame. The end lifter, `E_ML_L1`, allows a pallet to be transported to the lower conveyor frame from the end of the production line. The intermediate lifter, `M_ML_L1`, allows a pallet to be transported in both directions between the upper and lower conveyor frame. In addition, the intermediate lifter allows a pallet to traverse it, while remaining on the same conveyor frame.

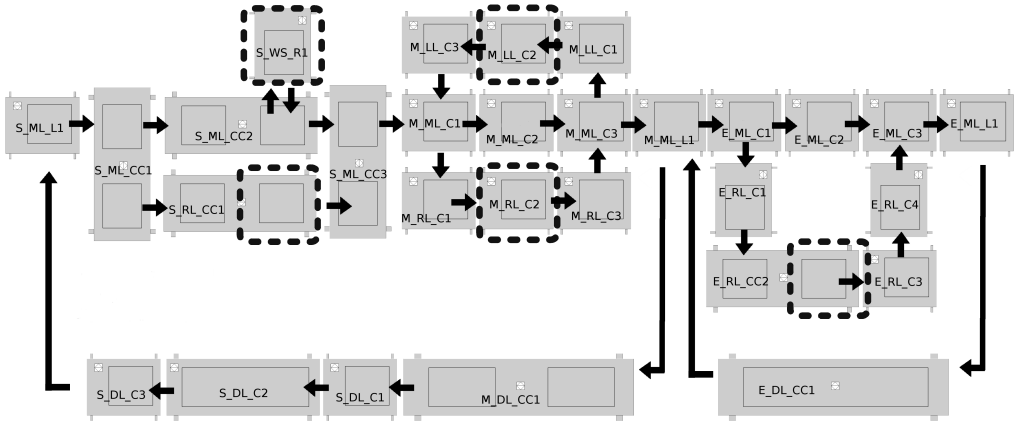
5.1.4 The Fastory Production System

The Fastory line consists of 12 robotic cells connected by a circular conveyor line. Each cell contains a conveyor consisting of five zones and a robot. The robot can perform assembly operations on any pallet that enters a certain conveyor zone inside the cell, which is called the robot processing location. In the test implementation, the assembly operations are in fact draw operations, in which the robot draws a mobile phone component onto the paper carried by the pallet. Thus, the papers represents the products being assembled. Figure 5.4 illustrates the layout of the simulated production line.

One of the robots, robot 1, is unable to perform assembly operations but can instead replace the paper carried by a pallet. To replace the paper, it first elevates



(a) The Socrades demonstration line consists of three portions sequentially connected.

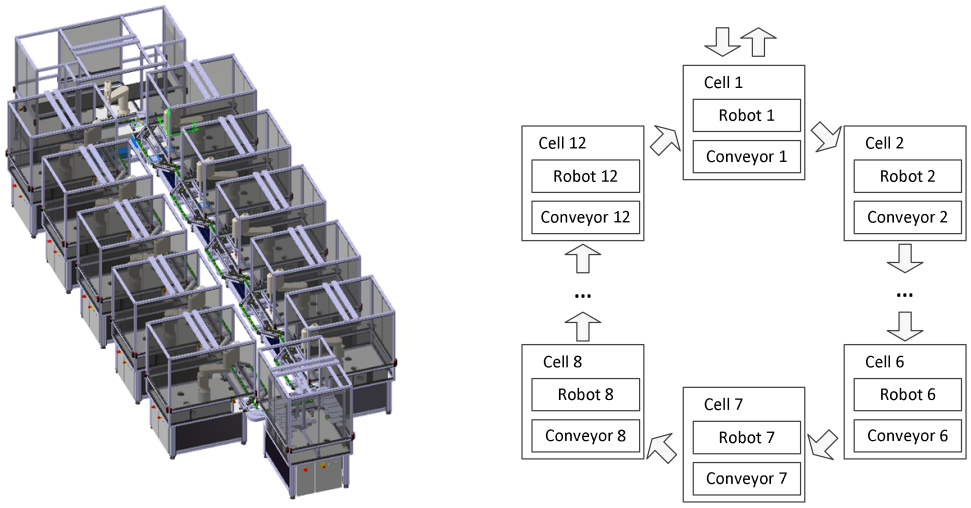


(b) A 2D overhead illustration of the upper and lower conveyor lines.

Figure 5.3: The demonstration line includes 29 conveyor segments, of which five are workstation processing locations.

the paper currently on the pallet and places it on the tray reserved for final products. Then, the robot removes a paper from the tray reserved for blank papers and places it onto the pallet.

Each cell includes a dedicated controller device for both the robot and the conveyor device. Each controller hosts a web service representing the controlled device. The service interfaces provide operations through which the devices can be controlled. While the interfaces for the robot and conveyor devices are significantly different, the interfaces of different conveyor and robot instances are identical between the 12 robotic cells. Only the set of operations in the robot 1 interface is somewhat different from the other robots. When the experiments described in this



(a) The robotic cells form a circle.

(b) Each robotic cell provides two web services.

Figure 5.4: The Fastory line consists of 12 robotic cells.

chapter were carried out, the conveyor cell 7 provided a direct route from cell 6 to cell 8. In particular, the cell included no functional robots but only a conveyor belt connecting the two adjacent cells.

The five conveyor zones in each robotic cell form two lanes. One of the lanes is traversed by pallets processed at the cell, whereas the other one is traversed by pallets bypassing the robot. Figure 5.5 illustrates the structure of the individual robotic cells in terms of the conveyor zones. Cell 1 differs from the other cells in that zone 4 is absent, and zone 2 is the robot processing location instead of zone 3.

The conveyor service interface includes the *Transfer* operation, which transports a pallet from one conveyor zone to an adjacent zone. In addition, the *TransferOut* operation makes it possible to transport pallets between adjacent robotic cells. Table 5.1 summarizes the operations supported by the conveyor service.

The final zone, zone 5, of each conveyor includes no pallet stopper. Hence, when a conveyor transports a pallet from zone 3 or zone 4 to zone 5, the pallet stops at zone 1 of the next conveyor on the line. However, one end of the pallet will still reside on the former conveyor. Therefore, before requesting the latter conveyor to move a pallet from zone 1 to zone 2 or zone 4, it is necessary to request the former conveyor to start running by invoking its *TransferOut* operation. The conveyor motor will remain active for a fixed time interval, and the service will send an *EquipmentChangeState* notification both at the beginning and end of the interval.

The robot service *Draw* operation performs an assembly operation specified by the input parameter on the pallet currently at the robot processing location. However, for robot 1, the *Draw* operation is substituted by the *ReplacePaper* operation, which replaces the assembly on the pallet with a new one and places the old one on the

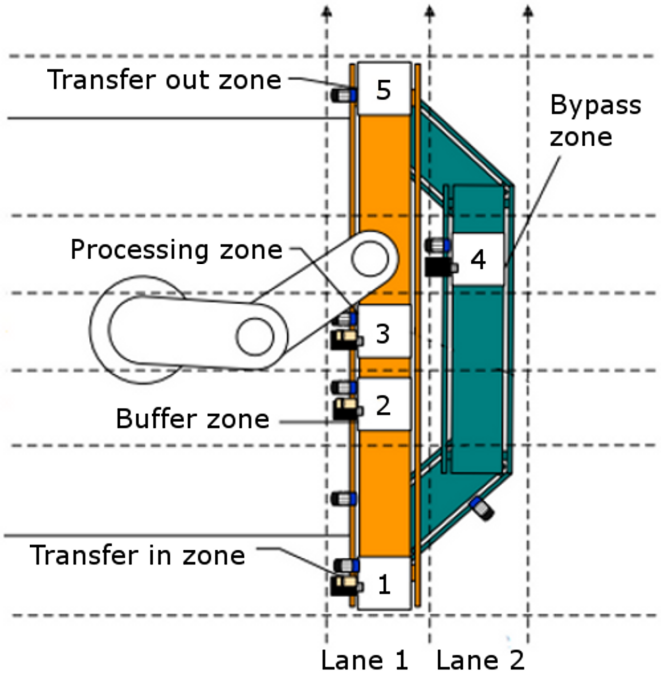


Figure 5.5: Each robotic cell contains five conveyor zones.

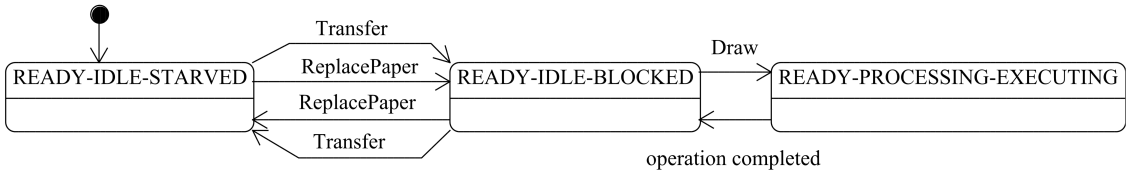


Figure 5.6: The Fastory robot service alternates between three states.

tray reserved for ready products. A robot can perform an operation only when a pallet occupies the robot processing location. In addition, each robot service sends the *EquipmentChangeState* notification whenever its internal state changes. For example, a notification is sent whenever the robot finishes an assembly operation. Table 5.2 summarizes the operations supported by the robot service.

The robot service is normally in one of three states: 'idle with no pallet', 'idle with a pallet', or 'performing an operation'. The three states are represented by the constant string values 'READY-IDLE-STARVED', 'READY-IDLE-BLOCKED' and 'READY-PROCESSING-EXECUTING'. The state diagram in Figure 5.6 illustrates the transitions between the states. In the figure, the WSDL operations that trigger the transitions are indicated by the labels attached to the transition arcs. Although the *Transfer* operation actually belongs to the conveyor service, it may still cause state changes in the robot service by transporting a pallet either into or out of the robot processing zone.

The web services constituting the Fastory line have been implemented both on

Table 5.1: The conveyor service provides three invokable operations and two event notifications.

Operation Name	**Type**	**Description**
Transfer	Request-response	Transfers a pallet between adjacent zones.
TransferOut	Request-response	Transports a pallet to the next cell.
GetState	Request-response	Lists the pallets occupying the zones.
TransferResultEvt	Notification	Indicates that a pallet has been successfully transferred between zones as a result of the *Transfer* operation.
PalletInEvt	Notification	Indicates that a pallet has arrived to zone 1 from the preceding cell.
EquipmentChangeState	Notification	Indicates that the conveyor has started or stopped unloading a pallet.

Table 5.2: The robot service provides two invokable operations and one event notification.

Operation Name	**Type**	**Description**
Draw	Request-response	Performs the requested assembly operation.
ReplacePaper	One-way	Replaces the paper carried by a pallet.
EquipmentChangeState	Notification	Indicates changes in the robot status.

S1000 controllers connected to actual devices and on Java applications simulating the devices. Application experiments involving the Fastory line are described in Sections 5.5, 5.6 and 5.7.

5.2 Applying BPEL in Simple Case Studies

This section presents an example of applying BPEL for service orchestration in a simple factory automation scenario. Meanwhile, several issues that can arise during the service orchestration are identified. The application scenario consists of a system of two sequential conveyors. The conveyors include a sensor near each end for detecting pallets. In the initial state, both of the conveyors are unoccupied. The goal is to transport a pallet through the two conveyors. The system is illustrated in Figure 5.7.

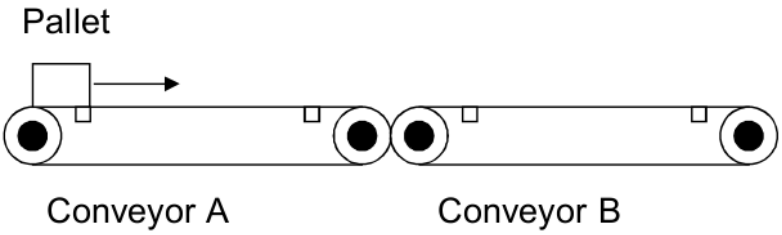


Figure 5.7: A system of two sequential conveyors.

Each of the conveyors provides a similar web service. In this example scenario, two of the operations supported by the service are relevant: *LoadFromLeft*, which transports a pallet onto the conveyor from the left and runs the conveyor until the pallet reaches the sensor near the right end, and *UnloadToRight*, which runs the conveyor to the right until the pallet has exited the conveyor. The operations are blocking, which means that they return a value only after they have completed. If an operation succeeds, it returns the status code 'SUCCESS'. The operation *LoadFromLeft* fails if the conveyor is already occupied by a pallet, in which case the operation returns the code 'BUSY', while the operation *UnloadToRight* fails if the conveyor is unoccupied, in which case the operation returns the code 'FAILURE'. A failure is typically detected immediately after an operation is invoked, and thus the blocking time is considerably shorter, as the actual transportation of the pallet is omitted.

Three different approaches to the orchestration of the system can be considered. Firstly, the conveyors may be started sequentially. Secondly, the conveyors may be started concurrently. Finally, the conveyors may communicate, so that conveyor A signals conveyor B to start running at the correct time.

If the conveyors are started sequentially, the composite service can be represented in BPEL using a single Sequence activity. The sequence contains four Invoke activities, which are executed in order. First, conveyor A loads and then unloads. Finally, the same actions are performed by conveyor B. Listing 5.1 shows an excerpt of the BPEL code that could be used in this case. The code has been generated by the NetBeans IDE from a workflow diagram created using the graphical editor. Since the activities are carried out sequentially, the two conveyors can only run individually and only as long as required to complete the services. In a highly simplified theoretical model, such a system appears to behave correctly. However, in practice, that only one conveyor is running at a time may prevent the pallet from moving from conveyor A to conveyor B. Because the latter remains stopped while the former is unloading, the pallet stops at the end of conveyor A. Finally, when conveyor B starts loading, the pallet remains on conveyor A.

```
<sequence>
      <receive name="Receive1" createInstance="yes"/>
      <invoke name="Invoke1" partnerLink="ConveyorA" operation="LoadFromLeft"
        xmlns:tns="http://www.example.org/ConveyorService"
        portType="tns:ConveyorService" outputVariable="Out"/>
      <invoke name="Invoke2" partnerLink="ConveyorA" operation="UnloadToRight"
        xmlns:tns="http://www.example.org/ConveyorService"
        portType="tns:ConveyorService" outputVariable="Out"/>
      <invoke name="Invoke3" partnerLink="ConveyorB" operation="LoadFromLeft"
        xmlns:tns="http://www.example.org/ConveyorService"
        portType="tns:ConveyorService" outputVariable="Out"/>
      <invoke name="Invoke4" partnerLink="ConveyorB" operation="UnloadToRight"
        xmlns:tns="http://www.example.org/ConveyorService"
        portType="tns:ConveyorService" outputVariable="Out"/>
</sequence>
```

Listing 5.1: The BPEL code for sequential conveyor operation.

The sequential starting of the operations can be achieved differently by using *links* inside a Flow activity. The Link construct is part of the WS-BPEL standard and makes it possible to specify dependencies between activities within an enclosing Flow activity. The complete operation can then be expressed using a Flow activity, which encloses four Invoke activities synchronized using three links, as illustrated in Figure 5.8. As in the case of using the Sequence activity, the operations provided by the conveyor service must be blocking. The first Invoke activity is declared the source of one of the links, and the second Invoke activity is declared the target for the same link. As each link has exactly one source and target, each Invoke activity is executed in order. The links must define such a sequence that conveyor A will first load and then unload, and finally conveyor B will perform the same activities. Unfortunately, this configuration includes exactly the same problem as using the Sequence activity: only one conveyor may run at a time. Thus, in practice the pallet would stop at the end of conveyor A. Moreover, as the Sun BPEL SE excludes links, the NetBeans IDE provides no support for creating a sequential invocation inside a Flow activity. Instead, the Sequence activity must be used. Nonetheless, in ActiveBPEL Designer, this appears the most straightforward approach to specifying a sequential invocation of operations.

To solve the problem with sequential invocation, where the transfer of a pallet from the first conveyor to the second fails, the two conveyors can be operated in parallel by using the Flow activity of BPEL. In this case, a Sequence activity is used as the main activity of the BPEL process. The first activity in the sequence, after the initiating Receive activity, is an invocation of the operation *LoadFromLeft* of conveyor A. The second activity is a Flow activity with two concurrent branches. One branch invokes the operation *UnloadToRight* of conveyor A, and one branch invokes *LoadFromLeft* of conveyor B. Thus, conveyor A will unload and conveyor B

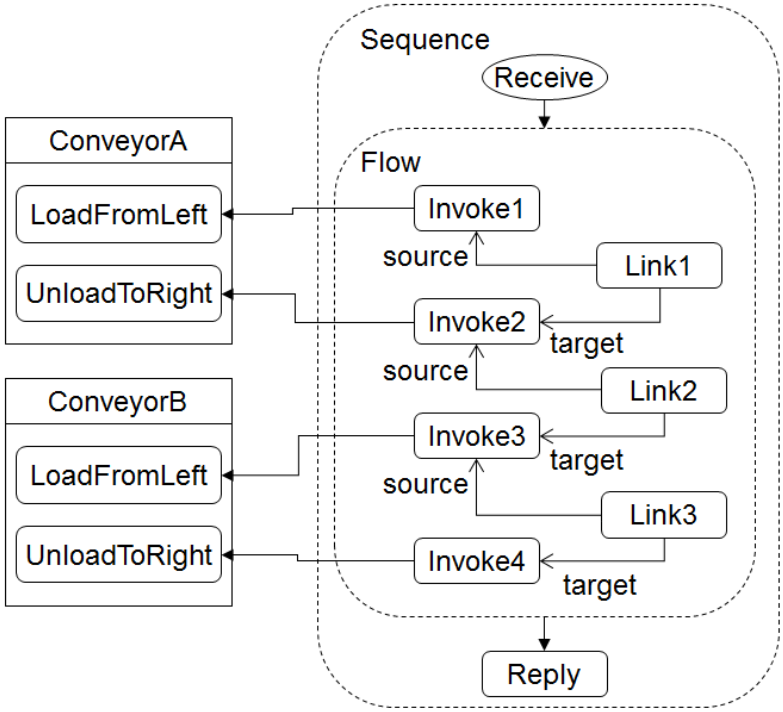


Figure 5.8: A BPEL process which uses links to achieve sequential invocation.

will load simultaneously. Thus, the pallet should be successfully transported from conveyor A to conveyor B, even if a more realistic physical model were used. The last Invoke activity in the sequence invokes the operation *UnloadToRight* of conveyor B. Figure 5.9 shows the BPEL process as a flow chart.

Perhaps a more complex approach would be to use a communication mechanism between the two instances of the conveyor service. The aim is that conveyor A is responsible for signaling conveyor B to start loading when A is starting to unload. This would ensure that neither conveyor runs longer than necessary and that the transportation of the pallet from one conveyor to the other succeeds.

Although the three BPEL tools discussed in Section 2.1 allow complex orchestration of web services, they appear inapplicable to presenting direct communication between web services, which is known as choreography. For example, ActiveBPEL Designer provides full support for WS-BPEL 2.0, which includes abstract processes. According to Pelz [75], abstract processes model web service choreography. However, after experimenting with ActiveBPEL Designer, it remains unclear how to address choreography using the tool. While WS-CDL (Web Services Choreography Description Language) [118] might be more applicable for this purpose, the reviewed tools provide no support for it. On the other hand, it is possible to orchestrate composite BPEL processes, as in fact each BPEL process models a web service which can be invoked by another BPEL process.

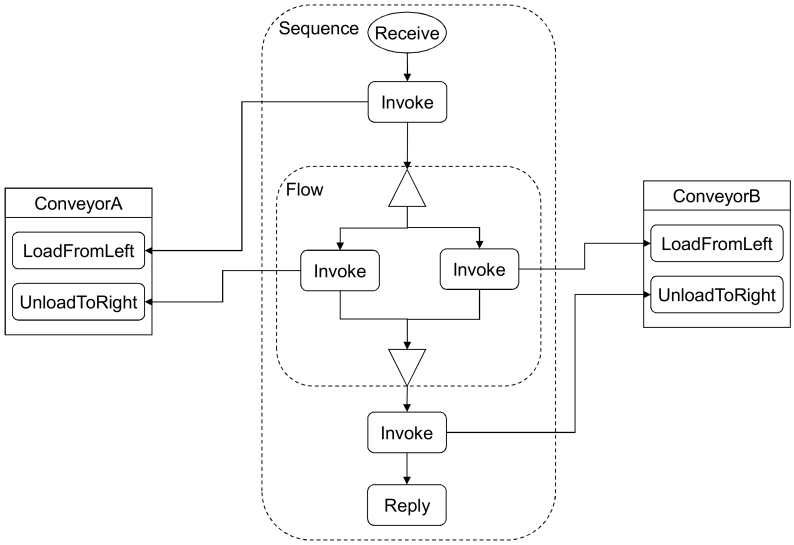


Figure 5.9: A BPEL process in which conveyor B loads a pallet while conveyor A is unloading.

5.2.1 Creating Composite BPEL Processes

To exemplify a composite BPEL process, the scenario in which conveyor A unloads and conveyor B loads concurrently is modeled with a composite BPEL process. The process is composed hierarchically so that it consists of two levels. The lower level involves two BPEL processes: one which models the loading and one which models the unloading of a conveyor. Each of these web services supports only one operation, which is named *load* or *unload*, respectively. The upper level contains only one BPEL process, which orchestrates the entire transport operation over the two conveyors. It is shown in Figure 5.10, which has been extracted from the NetBeans IDE.

The loading and unloading operations implemented by the two lower level BPEL processes may fail. However, the upper level BPEL process omits failure handling, as otherwise the process would be considerably more complex.

The upper level process invokes the services of four partner links. Two partner links are required for both conveyors: one for loading and one for unloading. The partner links use the WSDL files of the lower level BPEL processes. Thus, two web services are created from each of the two lower level BPEL processes. Each of the lower level BPEL processes uses directly the services of either conveyor A or conveyor B. In this case, the lower level processes are implemented as composite services; they do not rely on the *LoadFromLeft* and *UnloadToRight* operations of the conveyor service. Instead, they use the simpler operations, such as *SetMotorValue* and *SetMotorDirValue* to produce the same behavior. Therefore, the lower level BPEL processes are rather large, and this section omits detailed descriptions of

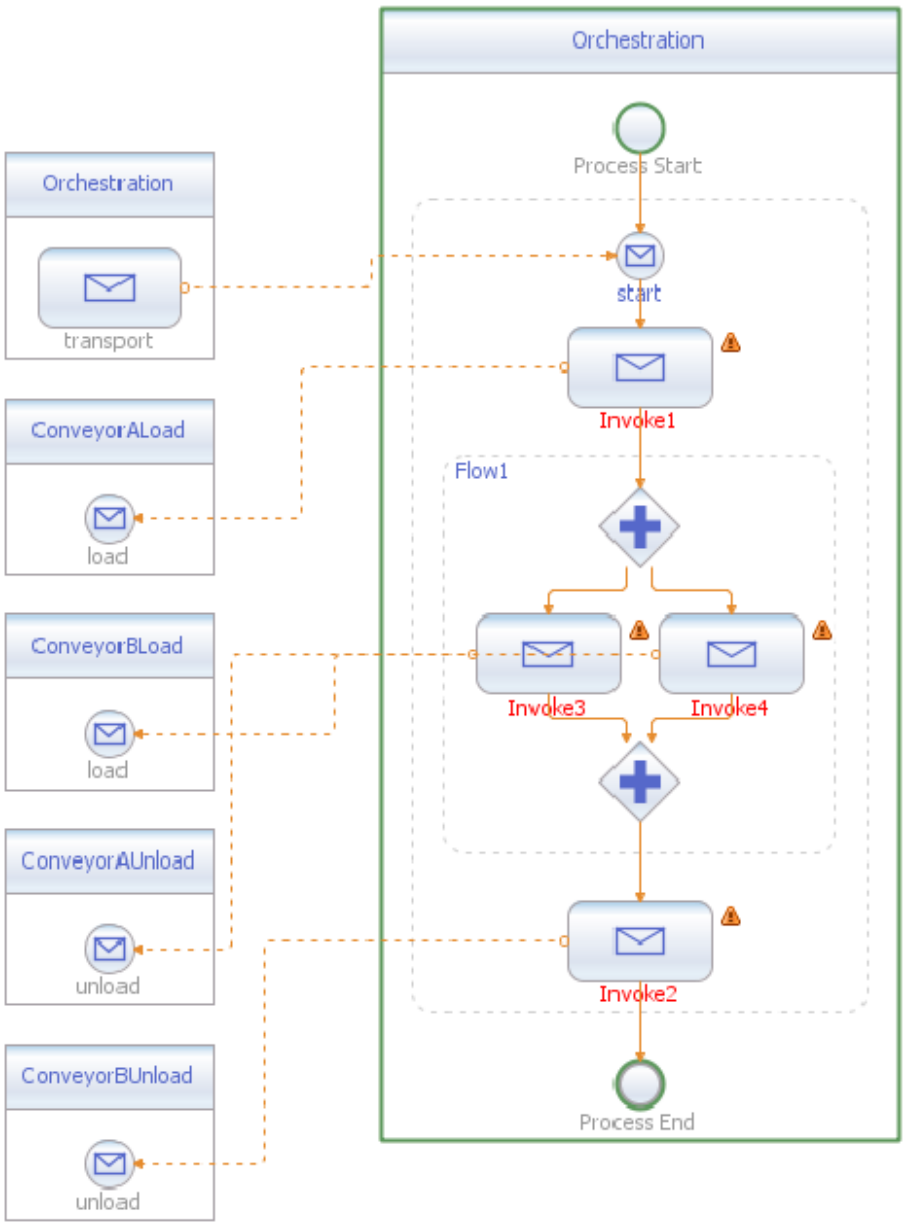


Figure 5.10: The upper level BPEL process in the composite service.

them.

Instantiating the upper level BPEL process results in a web service providing one WSDL operation. The operation is named *transport* and returns no response message. Invoking the operation results in the execution of the web service work flow defined in the BPEL process.

Especially in the factory automation domain, the lower level BPEL processes must frequently invoke operations on concrete physical devices rather than abstract web services. Hence, it is possible that a hardware error occurs and prevents a service from executing correctly, or that a device occasionally functions somewhat

slower than usual. This might cause that a service fails to complete at the normal time, resulting in indeterministic behavior. In these cases, the service is expected to respond with an appropriate error message. On the one hand, the event that a service requires a somewhat longer time to complete should typically pose no problems: The invoking service can wait until the invoked service has finished before continuing execution if synchronization is required. On the other hand, if strict real-time requirements are imposed on the system, it can do little to recover, even if it immediately detects the error. Hence, web services may be less applicable to production systems with such requirements. Yet, this is more a problem of web services in general than a problem of BPEL or the approach proposed in this dissertation.

5.2.2 Executing BPEL Processes Programmatically

Orchestration Tools includes Service Explorer, which is capable of executing BPEL files. Currently, the tool provides only partial support for the full WS-BPEL 2.0 specification; rather strict requirements have been set to the BPEL files that can be executed in Service Explorer.

When executing a BPEL process, Service Explorer treats the process more as a script than a web service. Thus, each execution of a BPEL process conceptually creates a temporary instance of the service represented by the BPEL process. The approach is different to the one employed by the BPEL tools discussed in Section 2.1, which create a web service instance of a BPEL process and deploy it on a server. Figure 5.11 illustrates how Service Explorer can be used for executing BPEL processes.

BPEL processes represent independent web services, but they frequently invoke the operations of other web services. References to other web services are made through partner links. When executing a BPEL process in Service Explorer, it is necessary that each partner link is associated to an actual device hosting a compatible service in the network. Thus, the user must select a device for each partner link in a dialog before executing a BPEL process. Different instances of a similar device type are identified by their serial numbers, which should typically be unique.

To associate devices to partner links, it is necessary that compatible devices have first been discovered. Service Explorer can discover devices in the network through the WS-Discovery protocol. Service Explorer considers only devices which host exactly one service. To obtain information on the discovered services, Service Explorer inquires the URLs to their WSDL files. Service Explorer can communicate with a web service only if a WSDL file can be obtained from the received URL or if the user specifies another valid WSDL file. However, the WSDL file must accurately correspond to the actual web service for the communication to succeed.

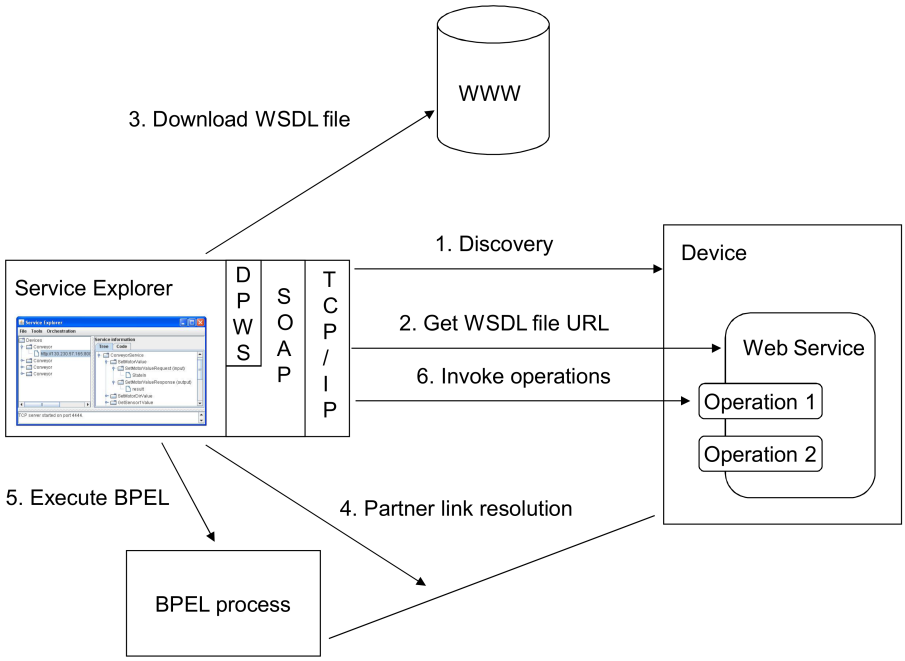


Figure 5.11: Using Service Explorer to execute BPEL files.

To determine whether a service hosted by a certain device is compatible with a certain partner link, the contents of the WSDL file describing the service are compared to the WSDL file specifying the requirements for the partner service in the partner link type definition. However, Service Explorer compares only a few properties between different WSDL files and may thus erroneously accept an incompatible service. If such a service is selected, Service Explorer may fail to execute the corresponding BPEL process.

As a case study, a system of four conveyors is considered. The conveyors are interconnected in a loop, as demonstrated in Figure 5.12. In the figure, the right end of each conveyor is marked with the letter R. To simplify this scenario, it is assumed that a conveyor can load a pallet from another conveyor using the same operations even if the conveyors are perpendicular.

A BPEL process which uses the conveyors to make the pallet travel a full cycle through the system can be swiftly created using NetBeans. Figure 5.13 shows a flow chart of the BPEL process with the initiating Receive and the ending Reply activities omitted. The final Assign activity simply assigns the string literal "DONE" to the output variable, which is returned to the invoker by the Reply activity. Again, error handling is omitted for the sake of simplicity.

The BPEL process illustrated in Figure 5.13 can be executed using the Service Explorer application. The execution of the process can be monitored by starting four instances of the conveyor client application, which can be used for displaying the status of a conveyor segment and invoking its operations.

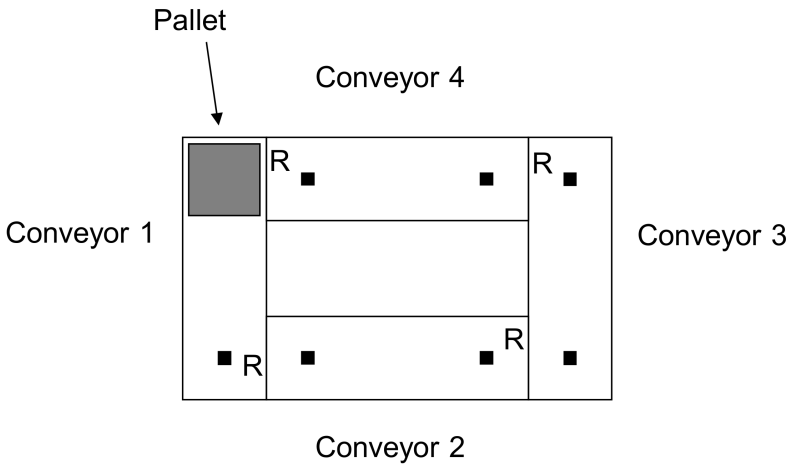


Figure 5.12: A loop of four conveyors, top-down view.

Figure 5.14 shows four conveyor client applications while Service Explorer is executing the BPEL process described above. The client windows, from left to right and top to bottom, represent the conveyors 1, 2, 3 and 4. The figure represents the moment at which conveyor 2 is unloading and conveyor 3 is loading. Thus, in the client applications, the pallet seems to appear on both conveyors for a short time.

The aforementioned loop of four conveyors can be implemented using a composite, or hierarchical, BPEL process as well. Then, the upper level BPEL process is otherwise similar to the one presented in Figure 5.13 except that eight partner links are needed instead of four. These partner links are of two types, one for unloading and one for loading a conveyor. Figure 5.15 illustrates the two BPEL processes that are used in the partner links. One instance of each type is needed for each of the four conveyors. Both processes include a similar partner link, which refers to an actual conveyor segment and is named 'Conveyor' in the figure.

When using a composite BPEL process, the user must specify in Service Explorer which of the partner links refer to a lower-level BPEL process instead of a device and which files contain the lower-level BPEL processes. In addition, the user must still specify for the lowest-level BPEL processes the devices to which the partner links refer. For example, even in this relatively simple scenario, eight partner links are used by the upper-level BPEL process, and each instance of the lower-level BPEL processes uses one partner link. Hence, a BPEL file must be specified for eight partner links and a device serial number for eight partner links. Currently, the entire task must be performed manually. Nevertheless, in the finalized system, this task will be automatically performed by an orchestration engine, which will determine the correct web service instance to perform a particular task at runtime [48].

Once the user has resolved the partner links in Service Explorer, the composite

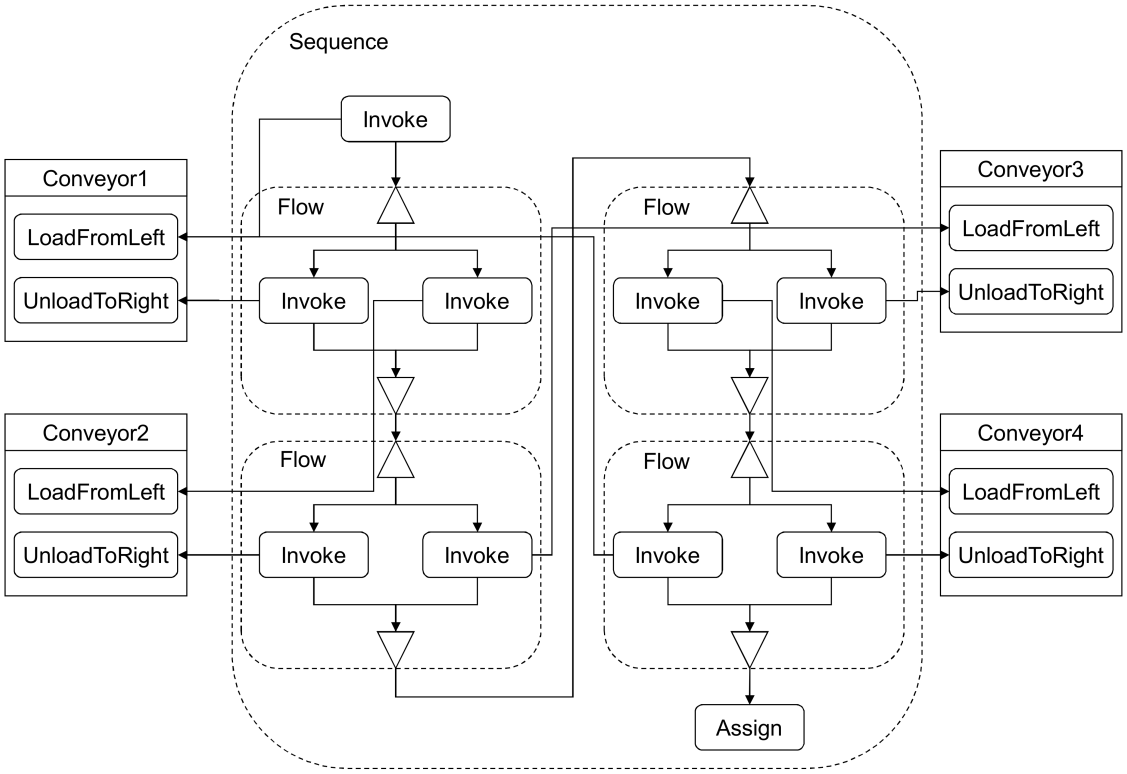


Figure 5.13: A BPEL process which operates a loop of four conveyors.

BPEL process can be executed. Each time the execution arrives to an Invoke activity which uses a partner link that refers to another BPEL process, the lower-level BPEL process is executed before continuing the execution of the upper-level BPEL process. As expected, the effects of running the composite process are exactly similar as when running the single-level BPEL process in Figure 5.13.

The number of partner links needed in the composite BPEL scenario could be reduced to four if the lower-level BPEL processes were able to perform both the loading and unloading operations. In standard BPEL, this can easily be achieved by using the Pick activity. The Pick activity may contain several branches of which only one is selected based on the received message [3]. Thus, the web service represented by a BPEL file can support several operations. However, the current version of Service Explorer is somewhat incomplete and provides no support for such use of the Pick activity.

Another way to reduce the number of partner links in the case of a composite BPEL process is to allow parameters to be passed to a BPEL process when it is executed. In this manner, the number of partner links can be reduced to four in the abovementioned scenario. Then, the lower-level BPEL processes perform either the loading from left or unloading to right, depending on the parameter value. Thus, only one lower-level BPEL process is required per conveyor segment. Figure 5.16

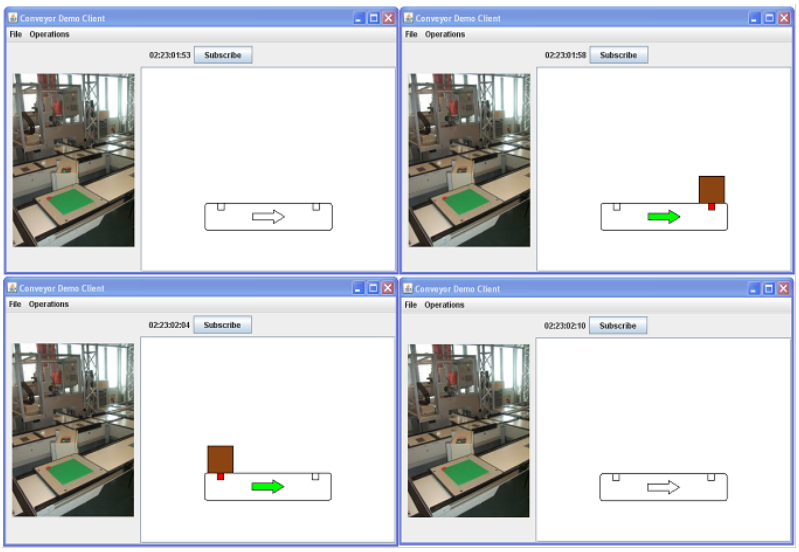


Figure 5.14: Four client applications monitoring the execution of a BPEL process.

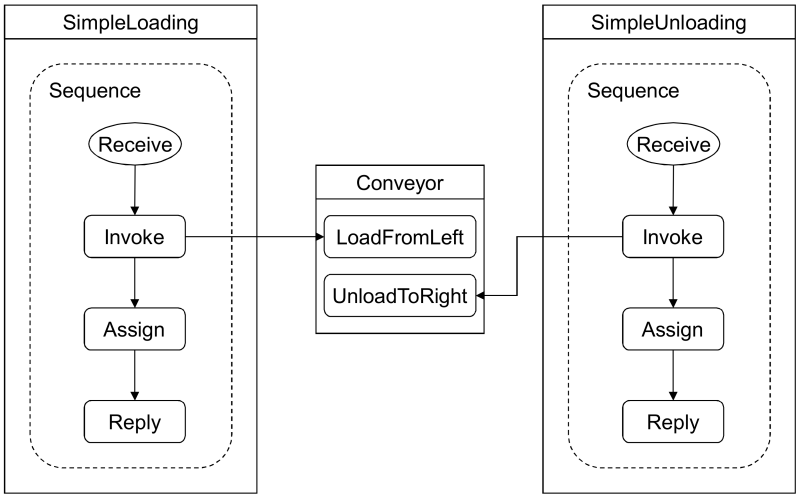


Figure 5.15: The lower-level BPEL processes used for unloading and loading.

shows the lower level BPEL process in this case. The initiating Receive activity accepts one input variable. If the value of the input variable is 'LOAD', then the operation *LoadFromLeft* is invoked on the conveyor segment. Otherwise the operation *UnloadToRight* is invoked. The upper-level BPEL process is otherwise similar to the one in Figure 5.13 except that the sequence begins with an Assign activity storing the correct values, 'LOAD' or 'UNLOAD', to the process variables subsequently provided as parameters to the lower-level BPEL processes. In addition, the partner link type used by the upper-level BPEL process is different in that it only contains one operation.

The approach to orchestrating web services using BPEL discussed in this dissertation may appear somewhat inflexible. For example, a BPEL process must be

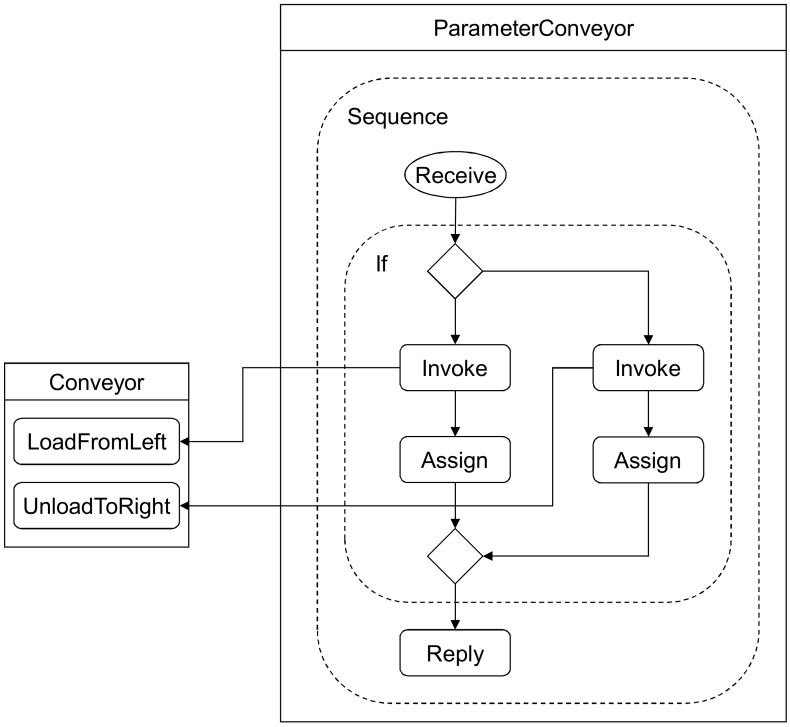


Figure 5.16: A BPEL process, which performs one of two operations based on a parameter value.

created in full before it can be executed. Moreover, a device requiring a specific web service is currently unable to, for example, query available services and select the most appropriate one at runtime. Instead, all interactions between web services must be known at the time of starting the execution of the BPEL process, and the endpoints to which the partner links refer must be specified. Nonetheless, in the finalized system, flexibility will be increased by the orchestration engine, which will automatically resolve partner links at runtime [48]. In addition, BPEL processes could probably be loaded from a server during runtime. Increasing the flexibility of the system to this level will require a considerable amount of further work.

5.3 Application Examples of Semantic Web Service Orchestration

This section presents two examples applying the web service orchestration approach proposed in Section 3.2. The first example involves only a single domain web service, whereas the second example involves several domain web services of the same type.

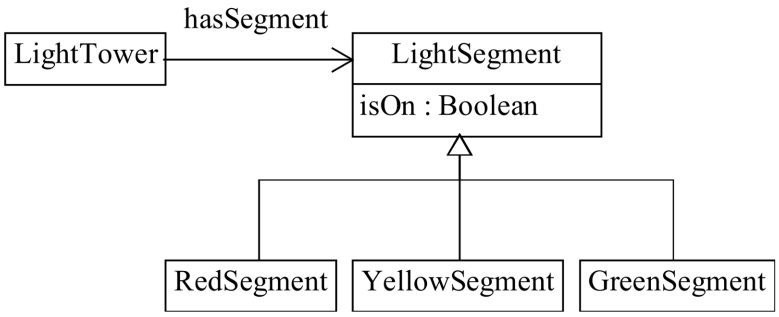


Figure 5.17: The main components of the light tower equipment ontology.

5.3.1 Light Tower Example

This experiment scenario describes the orchestration of a process using the domain web service representing a tower of three light segments described in Section 5.1.1.

The equipment ontology for the light tower web service contains a simple class hierarchy, which represents the three types of light segments, and the functional data type property *isOn*, which links each light segment to a Boolean value representing the power state of the segment. The object property *hasSegment* links the single *LightTower* individual to the three individuals representing the light segments. Figure 5.17 illustrates the contents of the equipment ontology.

The PC-based services are deployed by running separate Java applications. In addition, four orchestration support services are deployed: Ontology Manager, which updates the semantic model hosted by the Ontology Service, Orchestration Engine, which executes the BPEL process functioning as the orchestration instructions for this scenario, and Service Monitor, which assists the Orchestration Engine in initializing the partner links in the BPEL process.

The BPEL process developed for this experiment operates the light tower device so that it implements the logic of traffic lights. That is, the device continuously iterates through the four states typical to traffic lights in Finland. The BPEL process includes delays to add a fixed duration to each state. In particular, the state where only the red light is on continues for ten seconds, the state where both the yellow and red lights are on for three seconds, the state where only the green light is on for seven seconds, and the state where only the yellow light is on continues for three seconds, after which the system returns to the state where only the red light is on.

Before the Orchestration Engine begins executing the BPEL process, it uses Service Monitor to find appropriate matches for the partner links representing the aforementioned web services. The BPEL process first contains an Invoke activity to request Ontology Service to load the equipment ontology. Then, the process invokes

```
PREFIX lt:<http://www.ontologies.com/LightTower.owl#>
DELETE DATA
{
    lt:GreenSegment_1 lt:isOn false .
}
INSERT DATA
{
    lt:GreenSegment_1 lt:isOn true .
}
```

Listing 5.2: These SPARQL/Update operations modify the equipment ontology to reflect that the green light segment is lighted.

```
PREFIX tower: <http://www.ontologies.com/LightTower.owl#>
SELECT ?value
{
tower:RedSegment_1 tower:isOn ?value .
}
```

Listing 5.3: This SPARQL query retrieves the current state of the red light segment.

the *ScanNetwork* operation of Ontology Manager to cause Ontology Manager to subscribe to all event notifications sent by the light tower service. In addition, the process sets the update rules for Ontology Manager by invoking its *SetUpdateRules* operation and activates them by invoking the *SetActivityState* operation. In the light tower scenario, six update rules are required. While three of the rules are intended for reacting to the event that one of the light segments is activated, three are intended for reacting to the event that a segment is deactivated. Then, each time a light segment changes state, Ontology Manager receives a notification and requests the Ontology Service to update the equipment ontology accordingly. Each ontology update request contains a SPARQL/Update statement which the Ontology Service executes to modify the semantic model. For example, when the light tower service sends a notification message informing that the green segment has been lighted, Ontology Manager requests Ontology Service to execute the SPARQL/Update statement in Listing 5.2.

The BPEL process executed by Orchestration Engine invokes Ontology Service to determine the status of each light segment. It obtains the state information by requesting the Ontology Service to execute SPARQL SELECT queries such as the one in Listing 5.3. Then, the process requests the light tower service to activate or deactivate individual light segments so that the light tower enters the next traffic light state in the cycle.

Figure 5.18 shows a sequence diagram representing the interaction of the five web services in the experiment scenario. All of the components and relations in the figure directly correspond to the more general components in Figure 3.2. In

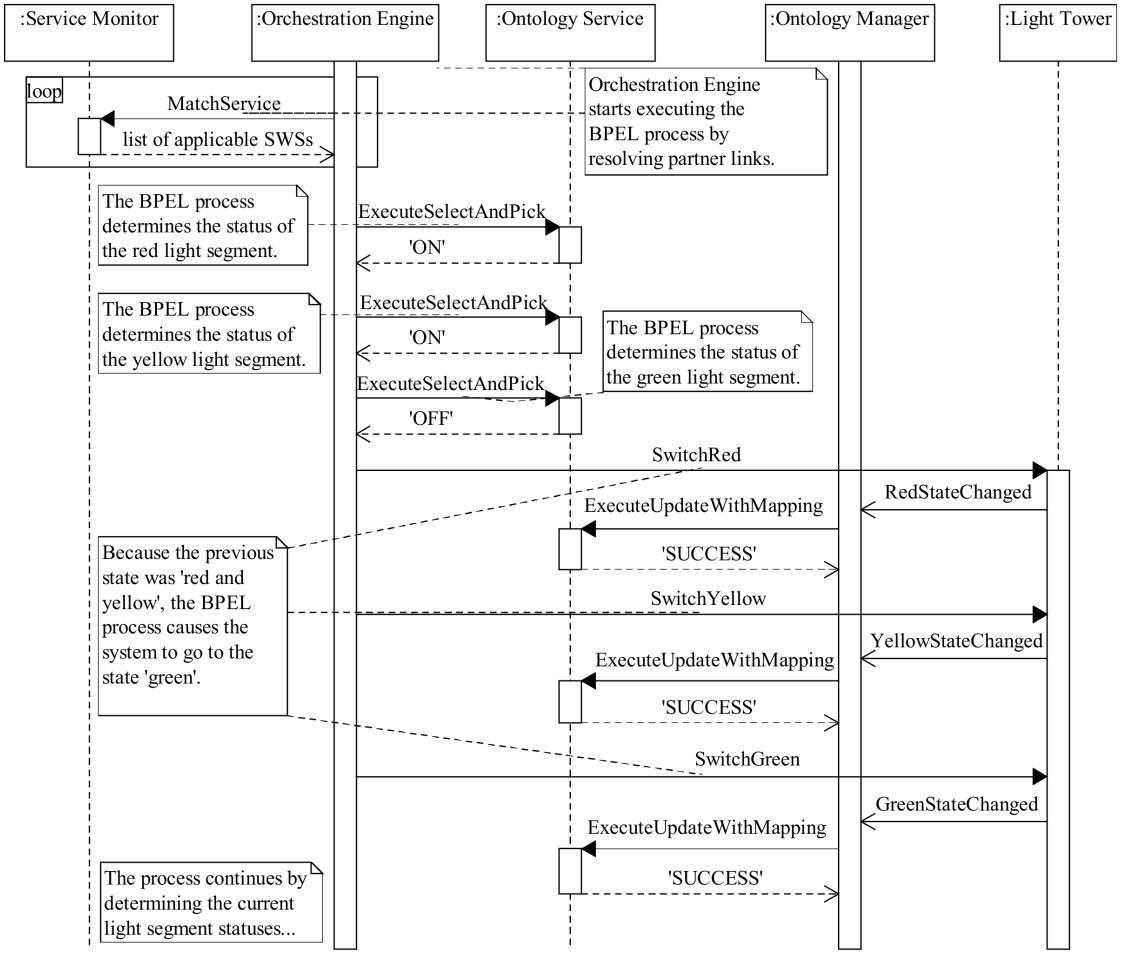


Figure 5.18: The light tower experiment sequence corresponds to the more general pattern depicted in Figure 3.2.

this example scenario, the light tower service is the only domain web service. The initialization of Ontology Service and Ontology Manager has been omitted from the diagram because it is performed similarly as in Figure 3.2.

Similar experiments have been performed using an alternative light tower web service interface. The alternative interface contains the composite operations *SetComplexStatus* and *ComplexStateChanged*. The former is a request-response type operation that makes it possible to set the states of all three light segments simultaneously, while the latter is an event notification sent by the service when the former operation has been invoked. The notification carries the new state information of all three light segments. The interface has been implemented for the light tower web service based on the S1000 controller. When the alternative interface is used, the BPEL process, which forms the execution logic of Orchestration Engine, can be modified to use the new composite request-response operation. In addition, the ontology update rules must be modified, and the WSDL description of the

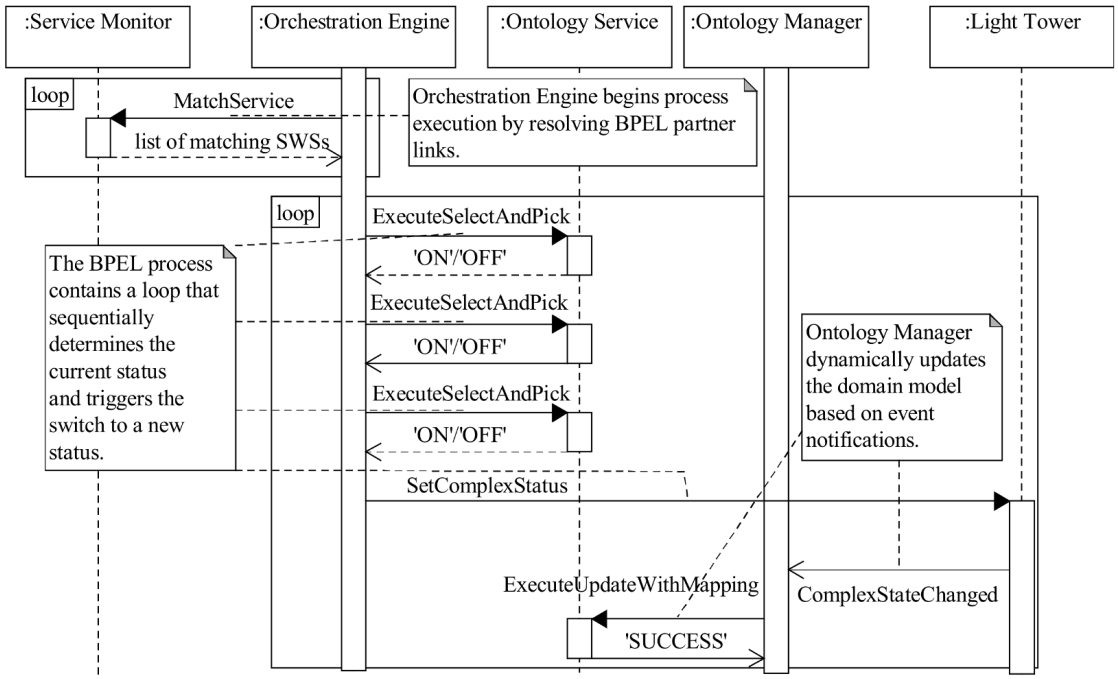


Figure 5.19: The BPEL process invokes fewer operations when using the alternative light tower service interface.

new interface must also be enriched with semantic information through the use of SAWSDL and OWL-S. Nonetheless, creating the semantic service description is less elaborate in this case, because the number of operations, and hence the number of generated OWL-S Process descriptions, is smaller. The sequence diagram of Figure 5.18 also applies to the alternative light tower service interface, except that to change the states of the light segments, only the *SetComplexStatus* operation needs to be invoked instead of the operations *SwitchRed*, *SwitchYellow* and *SwitchGreen*. In addition, only one notification is received after invoking the operation, namely *ComplexStateChanged*, instead of the notifications *RedStateChanged*, *YellowStateChanged* and *GreenStateChanged*. Ontology Manager compares the notification to the conditions of the update rules and sends update requests to Ontology Service. The scenario is illustrated in Figure 5.19.

5.3.2 Conveyor System Example

The conveyor system domain presented in Section 5.1.2 is more complex than the Light Tower system considered in Section 5.3.1. In particular, the former includes several web services representing conveyor devices, as opposed to a single web service representing a light tower device. In addition, each of the conveyor devices implements the service interface applied in Section 5.2. Nevertheless, the approach for orchestrating the services remains the same as in the previous subsection.

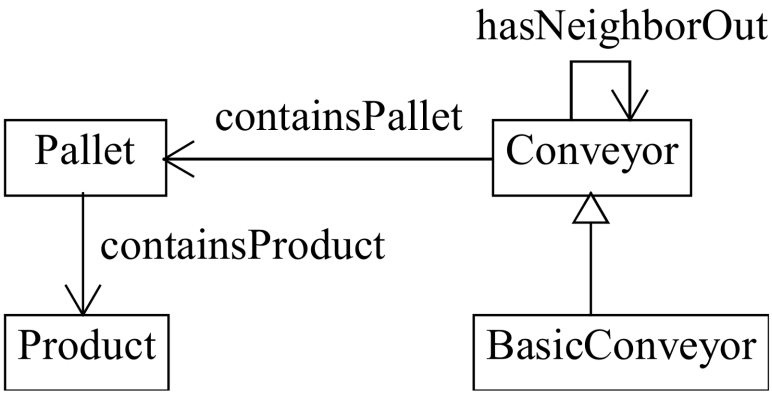


Figure 5.20: The main components of the conveyor system ontology.

Because the conveyors are part of the larger system presented in Section 5.1.3, this section will consider only a simplified equipment ontology, which includes only those parts of the overall system that are relevant for this example. Figure 5.20 shows a class diagram of the simplified ontology. The ontology contains six individuals of the class *BasicConveyor*, each of which represents one of the six conveyor segments. Because the conveyor segments are connected in a circle, the individuals in the ontology are connected in a loop by the *hasNeighborOut* property. Hence, a pallet may travel from conveyor A to conveyor B if and only if A is related to B by the *hasNeighborOut* property.

The conveyor system contains a pallet carrying a product, and the conveyor services are orchestrated so that the pallet traverses the conveyor loop a fixed number of times. The orchestration instructions are formulated as a BPEL process, which is then submitted to Orchestration Engine for execution.

The main part of the BPEL process consists of a cycle repeated for a fixed amount of times. In the beginning of each cycle, the Orchestration Engine sends a query to the Ontology Service to determine on which conveyor segment the product currently resides. If the product is on segment 5, then the next destination is segment 2. Otherwise, the next destination is segment 5. Orchestration Engine then queries from the Ontology Service the shortest path from the current location to the destination segment. The SPARQL query uses the GLEEN[2] library for requesting the shortest path. Once the shortest path has been determined, Orchestration Engine causes the pallet to traverse through the path by sequentially invoking the unloading and loading operations of each of the conveyor segments on the path. Invoking the operations causes the conveyor services to send event notifications, and Ontology Manager constantly sends update requests to Ontology Service so that the semantic

[2]GLEEN: Regular Paths for ARQ SparQL. Available at `http://sig.biostr.washington.edu/projects/ontviews/gleen/index.html`. Accessed on 2013-5-15.

model remains synchronized with the actual location of the pallet.

The BPEL process, which Orchestration Engine executes, begins with the initialization of the orchestration support services: first, Orchestration Engine invokes the *SetBaseOntology* operation on Ontology Service, as well as the operations *ScanNetwork*, *SetUpdateRules*, and *SetActivityState* on Ontology Manager. In addition, during each cycle, Orchestration Engine invokes the *ExecuteSelectAndPick* operation to determine the current pallet location and the *FindShortestPath* operation to determine the route to the next destination segment. After determining the path, Orchestration Engine moves the pallet by sequentially invoking the *LoadFromLeft* and *UnloadToRight* operations of the conveyor services representing the conveyor segments on the path.

Resolving the partner links is more problematic than in the Light Tower scenario because the number of domain web services is larger. In particular, because all of the domain web services are of the same type, differentiating between various instances of the same service type is crucial. Nonetheless, this can be achieved by creating a separate partner link type for each of the partner links representing the conveyor segments. The partner link type definitions are based on different WSDL files representing the conveyor service. The syntactic interface definition is similar in each file, but the port type SAWSDL annotations refer to different OWL-S Profiles. Thus, six instances of the conveyor service *profile* are created, each of which denotes a different conveyor segment. Then, to perfectly match a partner link, a conveyor service must have a port type SAWSDL annotation referring to the correct *profile* instance. Otherwise, the match rating will be lower. This allows Orchestration Engine to select the correct conveyor service instance for each partner link.

In this example, six update rules must be used. Each update rule is intended for reacting to the event that the pallet arrives on one of the conveyor segments. Each update rule contains a condition that matches the event that a pallet arrives on the latter sensor on the conveyor and an update statement which first erases any previous pallet location information from the equipment ontology and then sets the pallet location to the corresponding conveyor. In each update rule, the event source is given the same name as the partner link in the BPEL process, and the BPEL process contains an Invoke activity which reports the mapping between partner link names and endpoint URIs to Ontology Manager by invoking its *SetEndpointMapping* operation.

The notification *Sensor2Changed* is used in determining when a pallet arrives on the latter sensor of a conveyor. The notification is sent each time the sensor changes its state. The notification message contains the string ‘ON’ if the sensor detects a pallet, or ‘OFF’ if the sensor no more detects a pallet. Thus, if conveyor 3 sends a *Sensor2Changed* notification carrying the string ‘ON’, then the ontology

```
PREFIX ps: <http://www.owl-ontologies.com/ProductionSystem.owl#>
DELETE
{
    ?conveyor ps:containsPallet ?pallet .
}
WHERE
{
    ?conveyor a ps:BasicConveyor .
    ?pallet ps:containsProduct ps:productA .
}
INSERT
{
    ps:bcBasicConv3 ps:containsPallet ?pallet .
}
WHERE
{
    ?pallet ps:containsProduct ps:productA.
}
```

Listing 5.4: These SPARQL/Update operations set the location of the pallet to conveyor 3 in the equipment ontology.

is updated so that the new location of the pallet is conveyor 3. This is achieved by requesting the Ontology Service to execute the SPARQL/Update statement in Listing 5.4. The same statement both erases the previous location information of the pallet and writes the new information. This produces correct results because the system contains only one pallet. Basically, another update rule could be used to erase the previous information when a *Sensor2Changed* notification carrying the string 'OFF' was received. However, then the number of ontology update rules would be twice as large, which would require some additional work for performing the experiment. Moreover, if such a rule was used, then the equipment ontology would periodically reflect a state in which the pallet is on none of the conveyors. This is because the *LoadFromLeft* operation of a destination conveyor is not invoked until the *UnloadToRight* operation of the source conveyor has completed.

Figure 5.21 shows a sequence diagram representing the conveyor system experiment scenario. The six conveyor services are represented as a single object for the sake of simplicity. The diagram shows that once the Orchestration Engine has determined the path from the source conveyor to the destination conveyor, it starts repeating a cycle during which the pallet is moved to the next conveyor on the path. The loop is repeated until the pallet reaches the destination segment. The outer loop in the diagram corresponds to the BPEL *ForEach* activity that causes the pallet to repeat the loop a fixed number of times.

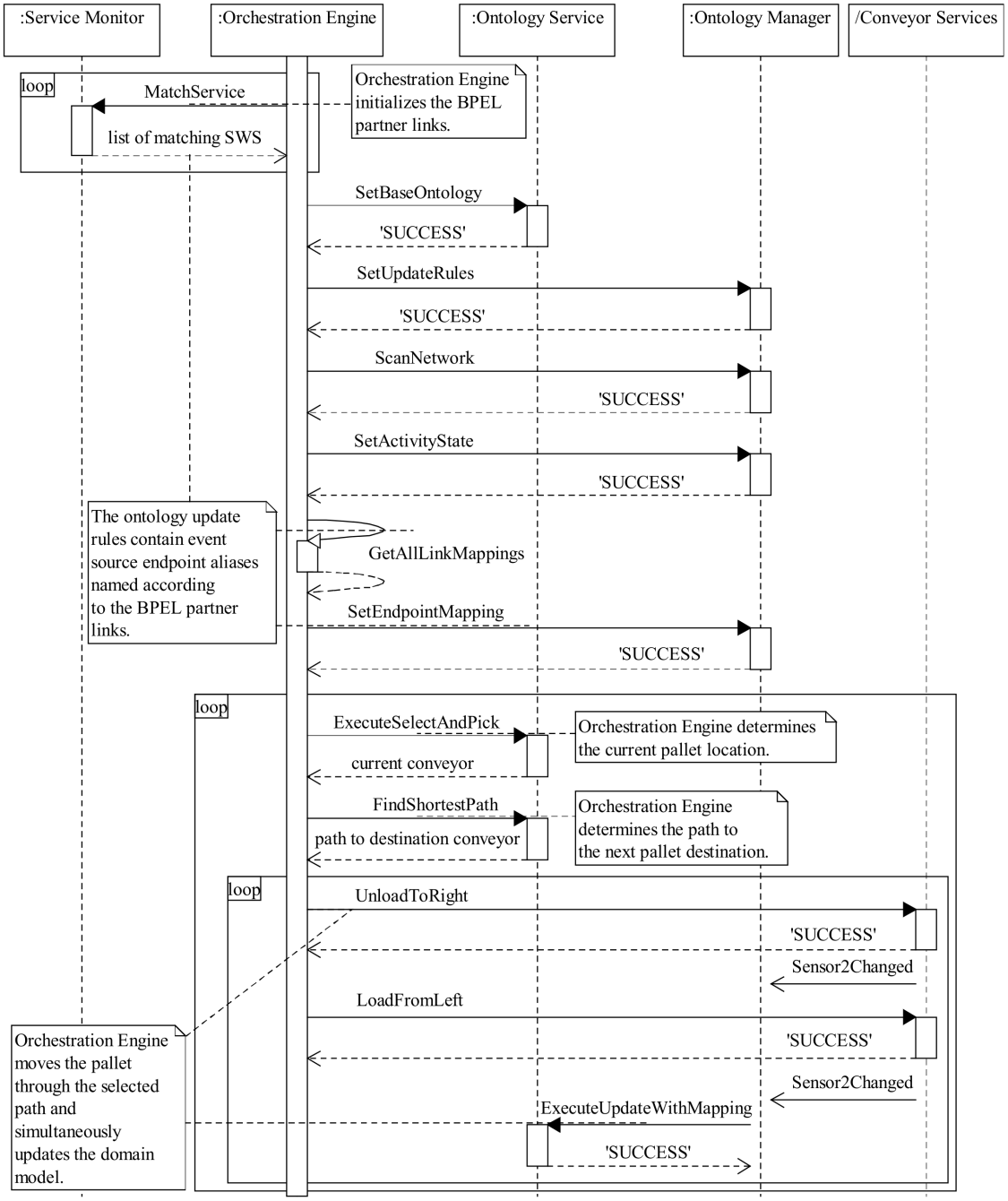


Figure 5.21: The BPEL process, which Orchestration Engine executes, uses Ontology Service to determine the path between the start and destination conveyor segments.

5.4 Application Example of SPARQL-based Semantic Web Service Composition

This section exemplifies web service composition in the domain presented in Section 5.1.3. The example applies the semantic web service composition approach presented in Section 3.3. The goal is to compose and execute a service that completely

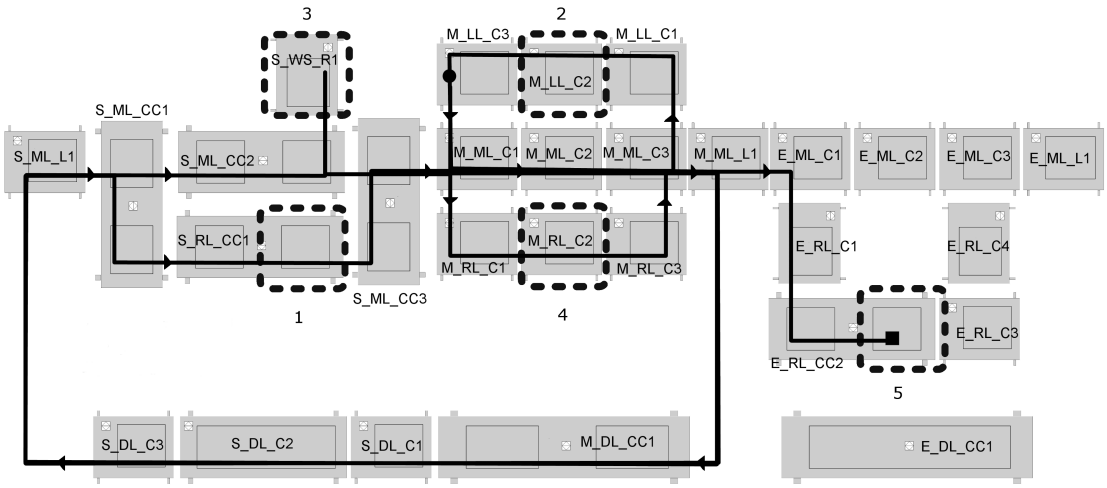


Figure 5.22: The path traversed by the pallet includes visiting each of the five workstations.

manufactures a product denoted by *product A*. The production sequence involves five operations performed on a pallet, and the two-dimensional map in Figure 5.22 indicates the path that the pallet must traverse to visit the necessary workstations. In the figure, the starting and ending points of the path are indicated by a dot and a rectangle, respectively. The numbers next to the workstations represent the order in which the pallet must visit each workstation.

The production line is described with a domain ontology including a total of 36 classes, and the UML class diagram of Figure 5.23 illustrates the most relevant concepts and relations in the ontology. While the domain ontology combines all of the information corresponding to the process, product and equipment ontologies described in [56], the product descriptions in particular are considerably simplified compared to those outlined in [56]. Each product requires a sequence of operations, and the class *Pair* makes it possible to represent the ternary relation between an operation sequence, an included operation, and a position in the sequence. In the domain ontology, the operation sequence performed on a product is associated with the pallet carrying the product. Therefore, instances of the class *Pallet* are associated with instances of the class *OperationSequence* through the property *hasReceivedProcessing*.

The modeled production system includes two machine categories: conveyor segments and workstations. Each of the devices is represented by a web service, which allows the devices to be controlled through web service interfaces. Hence, there are two types of web services. For the sake of simplicity, in the virtual model, all services of the same category are modeled using a similar web service. Only the semantic descriptions differ between the service instances.

The virtual model of the system consists of 34 web services simulating the be-

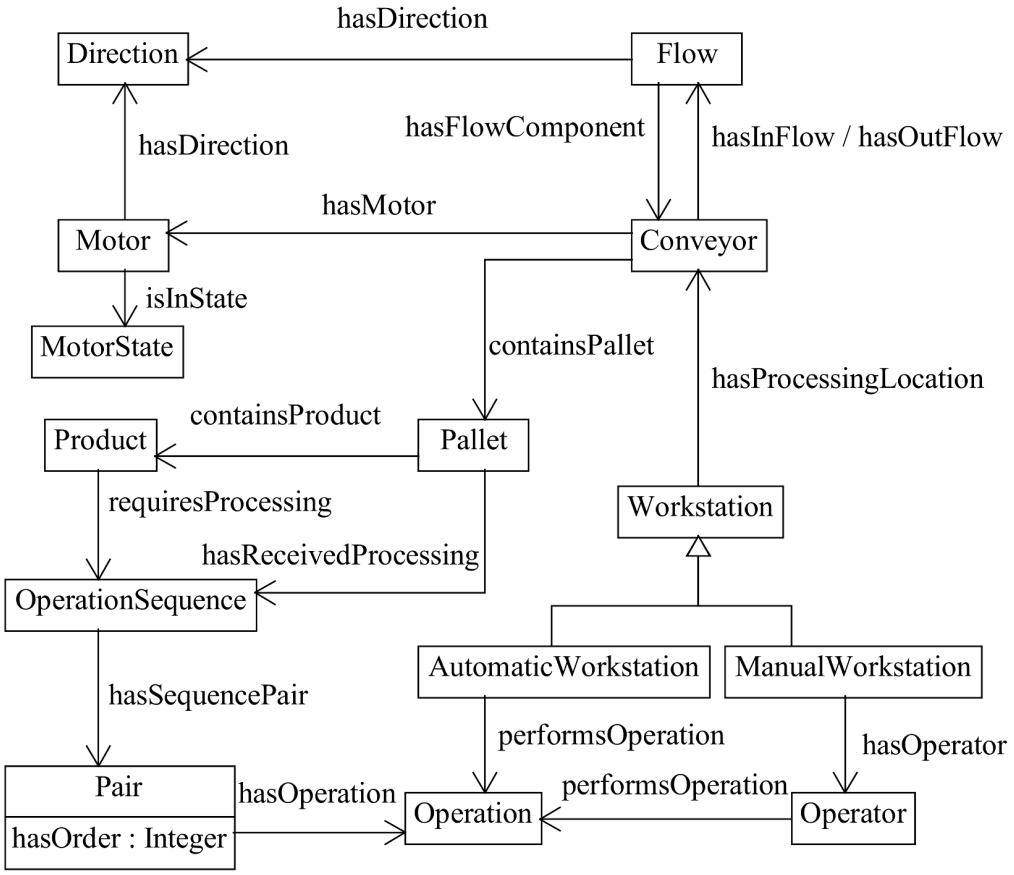


Figure 5.23: The domain ontology contains product, equipment, and process descriptions. All of the relations between the OWL classes are object properties, and class attributes represent datatype properties.

havior of the web services in the actual production line. The virtual web services are implemented using a DPWS tool kit [18] for Java programming language. Each of the services includes both a syntactic WSDL description and a semantic OWL-S description.

The conveyor web service supports four WSDL operations relevant to the application scenario: *TransferIn*, *TransferOutStart*, *TransferOutStop* and *StatusChanged*. The former three operations are request-response type operations, and hence, their SAWSDL annotations point to OWL-S atomic processes with both input and output parameters, while the latter operation is a notification operation, hence its SAWSDL annotation points to an OWL-S atomic process with output parameters only.

For example, unloading a pallet from one conveyor to an adjacent conveyor can be achieved by first invoking the *TransferOutStart* operation on the former conveyor, then the *TransferIn* operation on the latter conveyor, and finally invoking the *TransferOutStop* operation on the former conveyor.

The workstation web service includes only one request-response type operation,

PerformOperation, and one notification operation, *OperationCompleted*. Hence, the operations can be semantically described using only two OWL-S processes. However, each of the five workstations and 29 conveyor segments is described by a separate variant of the OWL-S description, and in total the semantic descriptions include 126 OWL-S processes. The service composition algorithms consider 92 of the processes as planning operators, while the remaining 34 represent event notifications signaling world state changes.

The following code examples focus on the web service representing the conveyor segment `M_LL_C2` in Figures 5.3(b) and 5.22. For the sake of simplicity, both the conveyor segment and the corresponding web service will henceforth be denoted as Conveyor 2.

Listing 5.5 shows the OWL-S process *TransferInProcess2*. The process corresponds to the WSDL operation *TransferIn* of Conveyor 2. Listing 5.5 includes only a partial view of the process including the conditions and effects of a successful execution of the process. The conditions and effects are specified as SPARQL and SPARQL/Update expressions embedded directly into the OWL-S document. The expressions refer to the OWL individual *bcBasicConv2*, which represents Conveyor 2 in the domain model. For representation purposes, the code listing includes extra line breaks, each of which is indicated by a pair of arrows.

Lines 10-16 in the condition expression of Listing 5.5 require that an initially loaded source conveyor must be unloading to Conveyor 2, while lines 17-19 require that the source conveyor is in the direction specified by the process input parameter *TransferInserviceIn2*, which is represented by a SPARQL query variable with the same name. The value of the variable denotes the direction from which Conveyor 2 should load a pallet. Finally, lines 20 and 21 require that Conveyor 2 is initially unoccupied.

In the effect expression of Listing 5.5, lines 30-32 state that the source conveyor no longer contains a pallet, while lines 33-35 state that Conveyor 2 contains a pallet after the execution of the OWL-S process. In addition, lines 36-41 specify that the pallet on Conveyor 2 is the one that was originally on the source conveyor.

The OWL-S process *StatusChangedProcess2*, which corresponds to the *StatusChanged* event notification of Conveyor 2, has two results. One of these signals the event that the conveyor motor state has changed, and the other additionally signals that a pallet has been transferred to the conveyor. The effect and condition expressions are quite similar to the those presented for *TransferInProcess2*. The WSDL notification operation refers to the corresponding OWL-S process through an SAWSDL annotation as illustrated in Listing 5.6.

The product, denoted as *product A*, requires five operations to be performed. As each of the operations is performed by a different workstation, the pallet car-

```
1  <process:AtomicProcess rdf:about="http://www.pe.tut.fi/fast/SocradesConveyors.owl#↩
       ↪ TransferInProcess2">
2    <process:hasResult>
3      <process:Result
4        rdf:about="http://www.pe.tut.fi/fast/SocradesConveyors.owl#↩
             ↪ IFTransferInSuccessResult2">
5        <process:inCondition>
6          <expr:SPARQL-Condition>
7            <expr:expressionData rdf:datatype="http://www.w3.org/2001/XMLSchema#↩
                 ↪ string">
8              <![CDATA[PREFIX ps:<http://www.owl-ontologies.com/↩
                   ↪ ProductionSystem.owl#>
9              ASK {
10             ?source ps:containsPallet ?pallet .
11             ?source ps:hasMotor ?motor .
12             ?motor ps:hasDirection ?outDirection .
13             ?source ps:hasOutFlow ?outFlow .
14             ?outFlow ps:hasDirection ?outDirection .
15             ?outFlow ps:hasFlowComponent ps:bcBasicConv2 .
16             ?motor ps:isInState ps:runningState .
17             ps:bcBasicConv2 ps:hasInFlow ?inFlow .
18             ?inFlow ps:hasDirection ?TransferInServiceIn2 .
19             ?inFlow ps:hasFlowComponent ?source .
20             OPTIONAL { ps:bcBasicConv2 ps:containsPallet ?palletB } .
21             FILTER (!BOUND(?palletB))
22             }]]>
23           </expr:expressionData>
24         </expr:SPARQL-Condition>
25       </process:inCondition>
26       <process:hasEffect>
27         <expr:SPARQL-Expression>
28           <expr:expressionData rdf:datatype="http://www.w3.org/2001/XMLSchema#↩
                 ↪ string">
29             <![CDATA[PREFIX ps:<http://www.owl-ontologies.com/↩
                   ↪ ProductionSystem.owl#>
30             DELETE {
31             ?inputConveyor ps:containsPallet ?pallet .
32             }
33             INSERT {
34             ps:bcBasicConv2 ps:containsPallet ?pallet .
35             }
36             WHERE {
37             ?inputConveyor ps:containsPallet ?pallet .
38             ps:bcBasicConv2 ps:hasInFlow ?inFlow .
39             ?inFlow ps:hasDirection ?TransferInServiceIn2 .
40             ?inFlow ps:hasFlowComponent ?inputConveyor .
41             }]]>
42           </expr:expressionData>
43         </expr:SPARQL-Expression>
44       </process:hasEffect>
45     </process:Result>
46   </process:hasResult>
47 </process:AtomicProcess>
```

Listing 5.5: The OWL-S description of Conveyor 2 *TransferIn* operation. Some alternative process results as well as inputs and outputs have been omitted for brevity.

```
<wsdl:operation name="StatusChanged">
    <sawsdl:attrExtensions    sawsdl:modelReference="http://www.pe.tut.fi/fast/↩
        ↪ SocradesConveyors.owl#StatusChangedProcess2" />
    <wsdl:output message="tns:IFStatusChangedServiceResponse" />
</wsdl:operation>
```

Listing 5.6: WSDL operations refer to OWL-S processes via SAWSDL annotations.

rying the product must visit each of the workstations in the production line once. Hence, although the domain ontology specifies the sequence of required operations, a planning process is necessary to determine the exact procedure for performing the operations on the product using the web services available.

To initiate the planning process, a BPEL process can be created to invoke the *FulfilGoal* operation of ServiceMonitor with a goal expression requiring all necessary operations to be performed on the pallet as well as a restriction expression specifying that at most one conveyor may be unloading at a time and that the sequence of performed operations must remain consistent with the required sequence in all intermediate domain states. The restrictions effectively reduce the state-space graph size and focus the search on only a few potential execution traces, thus allowing Algorithm 1 to succeed. The SPARQL query in Listing 5.7 represents the production goal, while Listing 5.8 expresses the aforementioned restrictions on the intermediate states.

In the goal query, lines 3-7 count the number of operations performed on the product, and lines 8-11 count the number of operations required to produce one unit of product A. Lines 12-22 count the number of matching entries in the sequence performed and the sequence required. Finally, line 27 uses the three computed values to specify that the sequence performed should be equal to the sequence required. In addition, lines 23-26 together with line 28 require that all conveyors are stopped in the goal state.

In the restriction query, lines 3-9 count the number of operations performed on the product, and lines 10-20 count the number of matches in the operation sequence required and the sequence performed. Lines 22-25 count the number of conveyors that are running, lines 26-28 determine the conveyor carrying the pallet, and lines 29-37 determine the conveyors that are unloading to the conveyor currently carrying the pallet. Finally, lines 39-41 use the computed values to require that the operation sequence performed is a beginning subsequence of the sequence required, that at most one conveyor motor may be active at a time, and that an active conveyor segment must be either the one currently holding the pallet or a neighboring upstream conveyor segment.

When Service Monitor initially receives a request to fulfill the goal, it composes a new composite OWL-S process with 128 web service invocations. The composed plan

```
1  PREFIX ps: <http://www.owl−ontologies.com/ProductionSystem.owl#>
2  ASK {
3  {SELECT (count(?pair) AS ?numPerformed) WHERE {
4  ?pallet ps:containsProduct ps:productA .
5  ?pallet ps:hasReceivedProcessing ?sequence .
6  ?sequence ps:hasSequencePair ?pair .
7  }}
8  {SELECT (count(?requirement) AS ?numRequired) WHERE {
9  ps:productA ps:requiresProcessing ?requirements .
10 ?requirements ps:hasSequencePair ?requirement .
11 }}
12 {SELECT (count(?pair) AS ?numMatches) WHERE {
13 ?pallet ps:containsProduct ps:productA .
14 ?pallet ps:hasReceivedProcessing ?sequence .
15 ?sequence ps:hasSequencePair ?pair .
16 ?pair ps:hasOperation ?performedOperation .
17 ?pair ps:hasOrder ?performedOrder .
18 ps:productA ps:requiresProcessing ?requirements .
19 ?requirements ps:hasSequencePair ?requirement .
20 ?requirement ps:hasOrder ?performedOrder .
21 ?requirement ps:hasOperation ?performedOperation .
22 }}
23 OPTIONAL {SELECT ?runningConveyor WHERE {
24 ?runningConveyor ps:hasMotor ?motor .
25 ?motor ps:isInState ps:runningState .
26 }}
27 FILTER (?numMatches = ?numRequired && ?numPerformed = ?numRequired
28 && !BOUND(?runningConveyor))
29 }
```

Listing 5.7: The goal query requires that product A is completed and all conveyors are stopped.

involves moving the pallet through the path depicted in Figure 5.22 and invoking the workstation services with correct input parameters. On a desktop computer with two 3 gigahertz CPU cores and 3.7 gigabytes of RAM, the associated planning process is completed in approximately 20 seconds. On subsequent requests, after the completed product has been removed from the system, Service Monitor may directly reuse the previously composed OWL-S process if the pallet is at the same initial location.

Service Monitor automatically composes the condition and effect expressions of a composed OWL-S process, which may cause them to be somewhat inaccurate. Therefore, whenever the solution plan discovered consists of a previously composed OWL-S process, Service Monitor simulates the plan execution by considering the condition and effect expressions of the atomic web service descriptions. Unless the simulated resulting world state satisfies the goal, Service Monitor attempts to determine a new solution plan without the previously composed process. For example, in the example scenario considered, Service Monitor simulates the execution result for

```
1  PREFIX ps: <http://www.owl-ontologies.com/ProductionSystem.owl#>
2  ASK {
3  {SELECT (count(?motor) AS ?runningCount) WHERE {
4  ?motor ps:isInState ps:runningState .}}
5  {SELECT (count(?pair) AS ?numPerformed) WHERE {
6  ?pallet ps:containsProduct ps:productA .
7  ?pallet ps:hasReceivedProcessing ?sequence .
8  ?sequence ps:hasSequencePair ?pair .
9  }}
10 {SELECT (count(?pair) AS ?numMatches) WHERE {
11 ?pallet ps:containsProduct ps:productA .
12 ?pallet ps:hasReceivedProcessing ?sequence .
13 ?sequence ps:hasSequencePair ?pair .
14 ?pair ps:hasOperation ?requiredOperation .
15 ?pair ps:hasOrder ?requiredOrder .
16 ps:productA ps:requiresProcessing ?requirements .
17 ?requirements ps:hasSequencePair ?requirement .
18 ?requirement ps:hasOrder ?requiredOrder .
19 ?requirement ps:hasOperation ?requiredOperation .
20 }}
21 OPTIONAL {
22 {SELECT ?runningConveyor WHERE {
23 ?motor ps:isInState ps:runningState .
24 ?runningConveyor ps:hasMotor ?motor .
25 }}
26 {SELECT ?palletLocation WHERE {
27 ?palletLocation ps:containsPallet ?pallet .
28 }}
29 {SELECT ?runningSourceConveyor WHERE {
30 ?runningSourceConveyor ps:hasMotor ?sourceMotor .
31 ?sourceMotor ps:isInState ps:runningState .
32 ?palletLocation ps:hasPallet ?pallet .
33 ?sourceMotor ps:hasDirection ?runningDirection .
34 ?runningSourceConveyor ps:hasOutFlow ?outFlow .
35 ?outFlow ps:hasDirection ?runningDirection .
36 ?outFlow ps:hasFlowComponent ?palletLocation .
37 }}
38 }
39 FILTER (?numPerformed = ?numMatches && ?runningCount <= 1
40 && (!BOUND(?runningConveyor) || ?runningConveyor = ?runningSourceConveyor
41 || ?runningConveyor = ?palletLocation))
42 }
```

Listing 5.8: The restriction query specifies that only one conveyor may unload at a time and that the operation sequence performed on product A must be consistent with the operation sequence required.

a plan consisting of 131 web service invocations in approximately 400 milliseconds.

5.5 Application Examples of Dynamic Domain Model Updating

The web service-based approach to hosting a domain model discussed in Section 3.4 is applied in [80] and [84]. In [80], the Ontology Manager based approach of updating the domain model is applied to maintain an accurate view of the manufacturing system while another web service, the *Orchestration Engine*, executes BPEL processes. In [84], the Service Monitor based approach is applied while Service Monitor plans and executes a process fulfilling a production goal.

This section presents an application example using the two approaches presented in Section 3.4 with the application domain described in Section 5.1.4. The prerequisite for both of the model update approaches is that an OWL model of the system

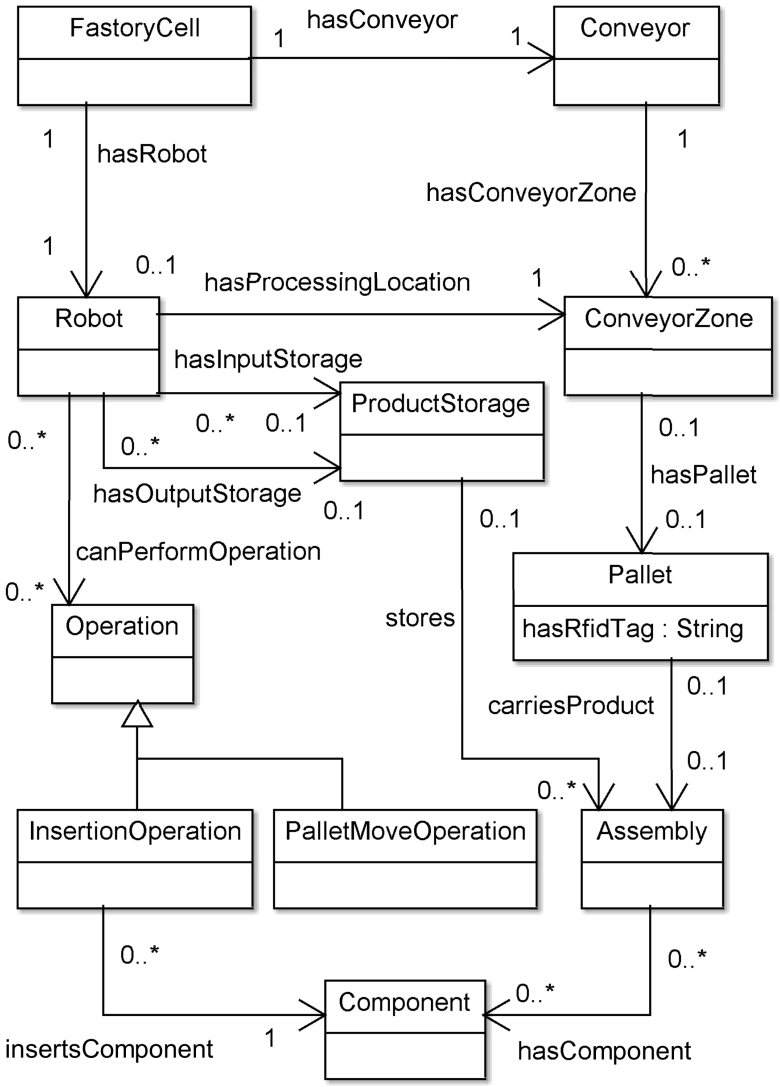


Figure 5.24: The manufacturing system ontology contains classes representing the production devices and product components.

is formulated. In the sequel, the OWL model used in these experiments will be described. The OWL domain model of the system consists of a generic production system ontology, which is designed to be applicable in various use cases, and a use case specific ontology, which extends the former to accurately model the particular production system considered. The use case specific OWL model is imported by a third OWL model which contains instance data for the modeled system. The OWL model containing the instance data will henceforth be called the ABox, while the OWL model containing only the static data will be called the TBox. The ABox contains all assertions of the current state of the system and is thus the only model that needs to be updated at run-time. Figure 5.24 shows a simplified diagram of the main concepts in the domain ontology.

The most integral concept in the domain ontology is the *Pallet* class, which represents the containers that carry the products being manufactured and are transported on the conveyor line. The carried product templates are represented by instances of the *Assembly* class. Each of the conveyors is composed of a number of zones, and each pallet can reside on one of the conveyor zones at a time. Hence, each instance of the *ConveyorZone* class is connected to at most one *Pallet* instance through the object property *hasPallet* in the OWL model.

Each product must at any time reside either on a pallet or a product storage. In the ABox, there are only two instances of *ProductStorage* class, which represent the two trays reserved for blank and completed product templates. Only the *Robot* instance representing robot 1 is connected to the *ProductStorage* instances through the OWL properties *hasInputStorage* and *hasOutputStorage*, as only robot 1 is able to retrieve empty product templates from the storage to a pallet occupying its processing location and deposit completed products from the pallet into storage.

5.5.1 Applying the Ontology Manager Approach

The Ontology Manager approach is designed to be applicable to traditional web services with no semantic descriptions. However, the update rules applied by Ontology Manager conceptually substitute a semantic description of the services. Therefore, the rules have to be manually created for each application scenario, such as the experiment scenario considered in this section.

Figure 5.25 shows the Ontology Manager GUI window with an update rule selected for editing. The selected rule, *Assembly Loaded Rule*, matches the event that the robot in cell number 1, or simply *robot 1*, retrieves a product template from the input storage onto an initially empty pallet. The tree in the left part of the window shows the required XML content for the notification message. The right part of the window shows the update actions to execute when the rule matches a notification. Furthermore, the rule defines one notification variable, *CellOutput*, whose value is

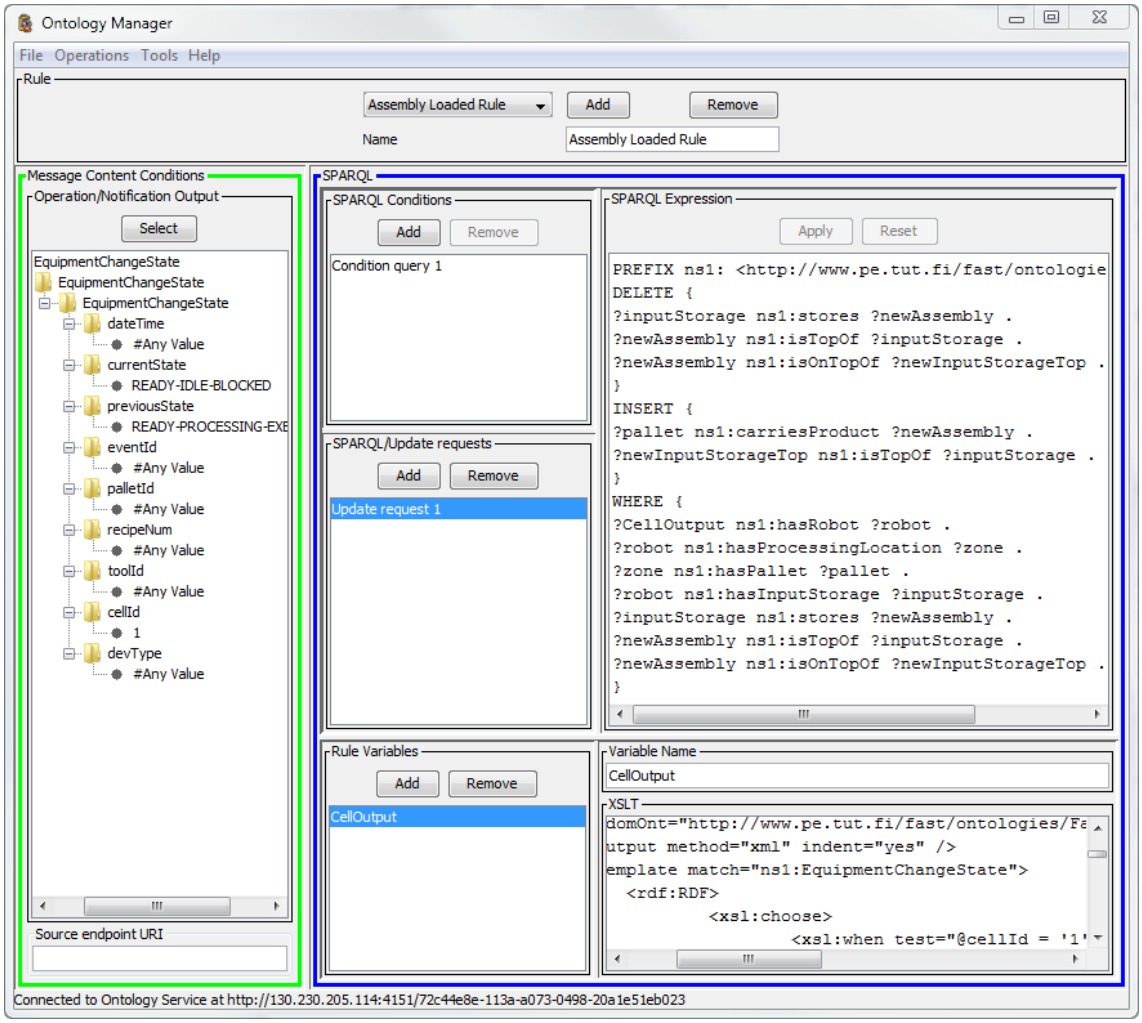


Figure 5.25: The Ontology Manager user interface facilitates the creation and editing of update rules.

extracted from the notification message content using the XSLT script in the lower right part of the window. The XSLT script extracts the value of the *cellId* attribute in the notification message content and translates it into the appropriate instance of the OWL class *FastoryCell*. The variable name and the translated value will then be sent to Ontology Service for use in executing the update actions. In addition, the rule contains one SPARQL ASK condition query, which will be included in the request message sent.

Two additional rules, *Assembly Switched Rule* and *Assembly Performed Rule* are necessary to accurately reflect the effects of the robot services on the domain state. While the rule names have no effect on rule processing in Ontology Manager, their main purpose is to allow a human user to interpret the rules.

Assembly Switched Rule corresponds to the event that the robot 1 service *replacePaper* operation is invoked when the pallet occupying the robot processing location already carries an assembly, whereas *Assembly Loaded Rule* corresponds to

Table 5.3: The effects of the robot services can be expressed with three update rules.

Rule Name	**SPARQL Condition**

Assembly Loaded Rule	PREFIX ns1: <http://www.pe.tut.fi/.../Fastory.owl#>
	ASK {
	?CellOutput ns1:hasRobot ?robot .
	?robot ns1:hasInputStorage ?inputStorage .
	?inputStorage ns1:stores ?newAssembly .
	?robot ns1:hasProcessingLocation ?zone .
	?zone ns1:hasPallet ?pallet .
	FILTER NOT EXISTS { ?pallet ns1:carriesProduct ?oldContent
	}
	}

Assembly Switched Rule	PREFIX ns1: <http://www.pe.tut.fi/.../Fastory.owl#>
	ASK {
	?CellOutput ns1:hasRobot ?robot .
	?robot ns1:hasProcessingLocation ?zone .
	?zone ns1:hasPallet ?pallet .
	?pallet ns1:carriesProduct ?oldContent .
	?robot ns1:hasInputStorage ?inputStorage .
	?inputStorage ns1:stores ?newAssembly .
	}

Assembly Performed Rule	PREFIX ns1: <http://www.pe.tut.fi/.../Fastory.owl#>
	ASK {
	?Operation a ns1:InsertionOperation .
	}

the event that the operation is invoked when the pallet is empty. Therefore, the condition and effect expressions are somewhat different between the two rules.

Assembly Performed Rule corresponds to the event that a robot completes an assembly operation, and the rule condition expression only specifies a restriction on the type of the operation performed.

The rule condition expressions ensure that each rule is applied in the correct domain state, even when the received event notifications alone are insufficient to indicate the correct rule to apply. Table 5.3 lists the three update rules and their condition expressions. For the sake of brevity, the table omits the SPARQL/Update expressions specifying the rule effects.

Once all rules have been specified by either creating them in the Ontology Manager GUI or by loading them from an XML file, Ontology Manager must be switched to an active state. The activation can be performed either locally by selecting the *Activate Rules* menu item in the *Tools* menu in Ontology Manager GUI or remotely by invoking the *SetActivityState* operation in the Ontology Manager web service

interface.

5.5.2 Applying the Service Monitor Approach

Service Monitor automatically detects the production system services as the devices hosting them are activated. If Service Monitor is deployed after the domain services, it discovers them during the automatic discovery process included in the Service Monitor initialization phase.

The WSDL documents published by the production system services specify no OWL-S processes. Instead, the WSDL operations include SAWSDL annotations referring to SWRL rules defined in the TBox. Therefore, when Service Monitor discovers, for example, the robot service, it generates an OWL ontology containing OWL-S processes that correspond to the robot WSDL operations. The OWL-S generation approach is based on SAWSDL annotations and its details are described in Section 3.5.

For all of the robot service instances except robot 1, the OWL-S process derived from the *EquipmentChangeState* WSDL notification operation includes three output parameters, which correspond to different components of the operation output message. Table 5.4 lists the output parameters, the types of their values, and their definitions.

Service Monitor derives the types for the OWL-S process parameters from the SAWSDL annotations of the XML schema type definitions used by the attributes in the WSDL operation output message. While the message schema includes a few other attributes, Service Monitor ignores them in the OWL-S derivation, since they contain no SAWSDL annotations.

While robot 1 provides no assembly operations, it enables the transport of product templates between the pallet occupying its processing location and the product template storages. Therefore, the XML schema in the robot 1 WSDL document contains no attribute indicating the performed operation, and the OWL-S process derived from the notification operation includes only the first two of the outputs

Table 5.4: The *EquipmentChangeState* OWL-S process has three output parameters.

Output Name	**Type**	**Description**
EquipmentChangeStateOutput1	RobotStatus (OWL)	The current robot status.
EquipmentChangeStateOutput2	RobotStatus (OWL)	The previous robot status.
EquipmentChangeStateOutput3	Operation (OWL)	The performed operation.

listed in Table 5.4.

Once Service Monitor has derived the OWL-S descriptions for the discovered robot services, it extracts domain model update rules from the OWL-S processes representing the robot *EquipmentChangeState* WSDL notification operations. Thus, for the robot 1 service, Service Monitor extracts one update rule with two conditional effects and two rule variables. The condition and effect expression language used in the OWL-S descriptions may be either SPARQL or SWRL, depending on the current Service Monitor settings. In the former case, the expressions are all SPARQL ASK queries, and therefore Service Monitor directly copies the condition expressions into the rule objects and translates the effect expressions to SPARQL/Update expressions. In the sequel, however, it is assumed that Service Monitor has been configured to generate SWRL expressions.

Table 5.5 shows the condition and effect expression for the OWL-S process result representing the case that robot 1 unloads the current product template from the pallet to the output storage and loads a new template from the input storage onto the pallet. While the first table column shows the SWRL condition expression on the first row and the effect on the second row, the second table column shows the SPARQL expressions to which Service Monitor translates the condition and effect during the extraction of the update rule object models. While the SWRL expressions embedded into the OWL-S documents are expressed in XML, the table shows them as plain text for the sake of brevity. The symbol namespaces are omitted in the table except for the *swrlb* namespace prefix denoting the namespace of SWRL built-ins. The condition and effect expressions refer to the output parameters and the corresponding notification variables by name.

Figure 5.26 shows the Service Monitor GUI displaying goal process statuses. In the figure, only one goal process has been created, and it is currently running. The top part of the GUI window displays the number of semantic web services discovered. In the situation displayed, the number includes the 22 domain web services as well as Ontology Service and Service Monitor itself. The lower left part of the window shows the SPARQL ASK query representing the goal to be achieved. Service Monitor has composed a solution plan consisting of a sequence of 43 web service operation invocations, of which it has already invoked the first seven. While Service Monitor executes the plan, it monitors the event notifications sent by the domain services and updates the domain model accordingly. Service Monitor proceeds to the next operation invocation in the sequence only when it detects that the previous operation has caused the expected effects on the domain model.

As the update rules used by Service Monitor are automatically derived and uneditable, there is no need to explicitly activate Service Monitor similarly to Ontology Manager. Instead, Service Monitor is typically in an active state and reacts to event

Table 5.5: Service Monitor converts result conditions and effects from SWRL to SPARQL.

SWRL Expression	**SPARQL Expression**

hasProcessingLocation(robot1, ?zone) ∧ carriesProduct(?pallet, ?assembly) ∧ hasPallet(?zone, ?pallet) ∧ canPerformOperation(robot1, ?operation) ∧ hasInputStorage(robot1, ?inputStack) ∧ isTopOf(?oldInputTop, ?inputStack) ∧ isOnTopOf(?oldInputTop, ?newInputTop) ∧ stores(?inputStack, ?oldInputTop) ∧ hasOutputStorage(robot1, ?storage) ∧ isTopOf(?oldTop, ?storage) ∧ PalletMoveOperation(?operation) ∧ sameIndividual(?EquipmentChangeStateOutput1, blocked) ∧ sameIndividual(?EquipmentChangeStateOutput2, executing)	ASK {
	robot1 hasProcessingLocation ?zone .
	?pallet carriesProduct ?assembly .
	?zone hasPallet ?pallet .
	robot1 canPerformOperation ?operation .
	robot1 hasOutputStorage ?storage .
	?oldTop isTopOf ?storage .
	robot1 hasInputStorage ?inputStack .
	?newAssembly isTopOf ?inputStack .
	?newAssembly isOnTopOf ?newInputTop .
	?inputStack stores ?newAssembly .
	?operation a PalletMoveOperation .
	FILTER (?EquipmentChangeStateOutput1 = blocked && ?EquipmentChangeStateOutput2 = executing)
	}

swrlb:remove(1) ∧ swrlb:remove(5) ∧ swrlb:remove(6) ∧ swrlb:remove(7) ∧ swrlb:remove(9) ∧ stores(?storage, ?assembly) ∧ isTopOf(?assembly, ?storage) ∧ isOnTopOf(?assembly, ?oldTop) ∧ carriesProduct(?pallet, ?oldInputTop) ∧ isTopOf(?newInputTop, ?inputStack)	DELETE {
	?pallet carriesProduct ?assembly .
	?oldTop isTopOf ?storage .
	?newAssembly isTopOf ?inputStack .
	?newAssembly isOnTopOf ?newInputTop .
	?inputStack stores ?newAssembly .
	}
	INSERT {
	?storage stores ?assembly .
	?assembly isTopOf ?storage .
	?assembly isOnTopOf ?oldTop .
	?pallet carriesProduct ?newAssembly .
	?newInputTop isTopOf ?inputStack .
	}
	WHERE {
	(The same as the query body in the above ASK query.)
	}

notifications from the domain services by applying the corresponding update rules.

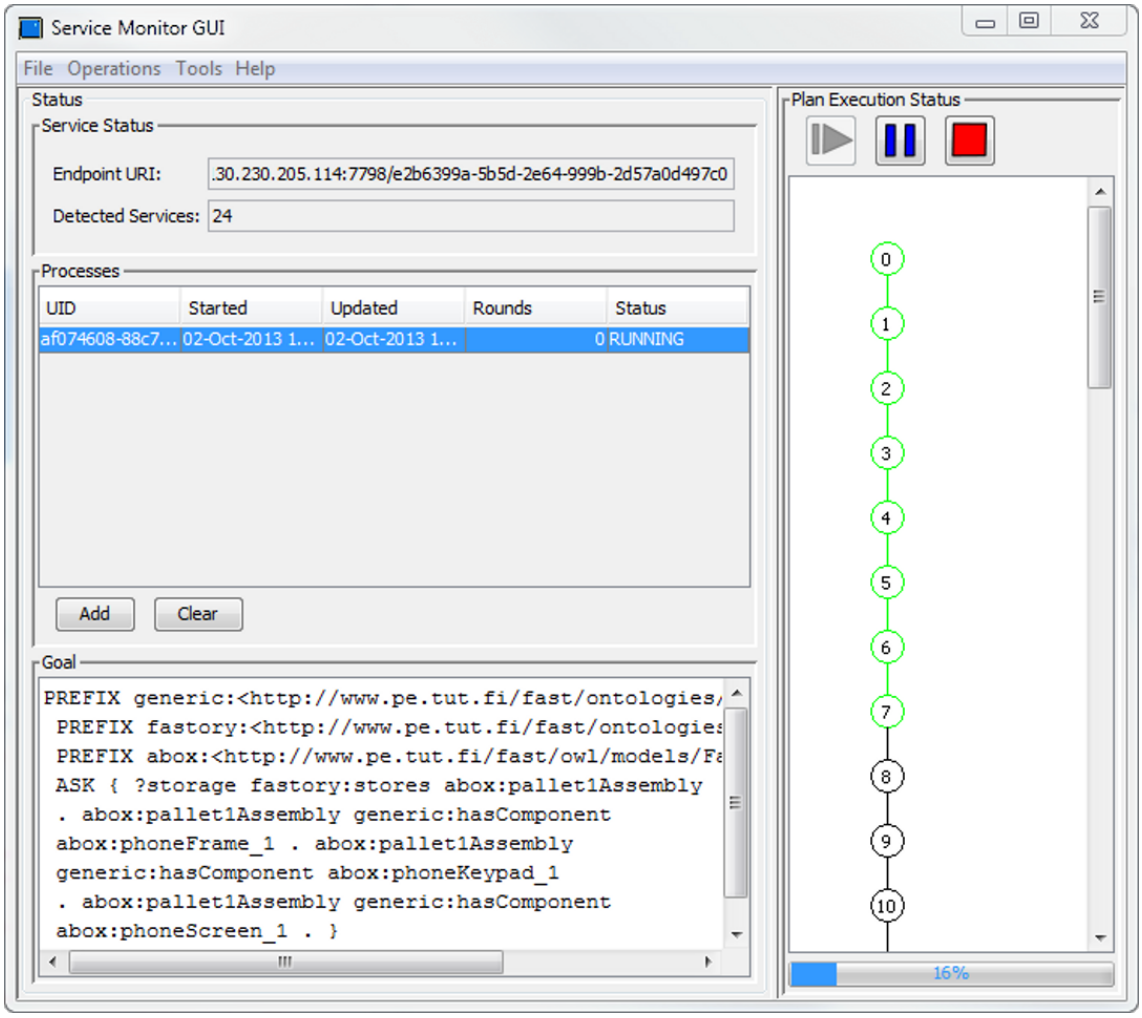


Figure 5.26: Service Monitor simultaneously applies ontology update rules during domain web service invocation.

5.6 Application Example of OWL-S Generation

This section exemplifies the application of the OWL-S derivation approach proposed in Section 3.5.

The current implementation of the Service Monitor web service has been tested both on PC-based web services implemented with Java and on web services implemented on RTUs controlling the actual system described in Section 5.1.4. The system is modeled by the OWL model described at the beginning of Section 5.5. The OWL model contains individuals representing the system components, such as the individual cells, robots, and conveyors.

Listing 5.9 shows the WSDL document describing the robot 2 service. The listing has been abbreviated so that it only contains the most relevant parts, and some parts and namespace URIs have been substituted with three dots. The WSDL operation *replacePaper*, which is present only in the robot 1 WSDL document and

```
<wsdl:definitions name="RobComm1"...>
  <wsdl:types>
      <xs:schema targetNamespace="http://www.tut.fi/fast/robot"...>
          <xs:element name="String" type="xs:string" />
          <xs:element name="calibrateRobot" type="xs:string"/>
          <xs:element name = "EquipmentChangeState">
              <xs:complexType>
                  <xs:attribute name = "currentState" use = "required" type="tns:DeviceState" />
                  <xs:attribute name = "previousState" use = "required" type="tns:DeviceState" />...
              </xs:complexType>
          </xs:element>
          <xs:simpleType name="DeviceState"
      sawsdl:modelReference="...#RobotStatus">
      <xs:restriction base="xs:string">
        <xs:enumeration value="READY-IDLE-BLOCKED"
          sawsdl:modelReference="...#blocked" />...
      </xs:restriction>
    </xs:simpleType>
          <xs:simpleType name="RecipeType"
      sawsdl:modelReference="...#Operation">
      <xs:restriction base="xs:string">
        <xs:enumeration value="1" sawsdl:modelReference="...#frameInsertion1" />...
      </xs:restriction>
    </xs:simpleType>
    <xs:element name="DrawMsg">
      <xs:complexType>
        <xs:sequence>
          ... <xs:element name="recipeNum" type="tns:RecipeType"/>...
        </xs:sequence>
      </xs:complexType>
    </xs:element>
      </xs:schema>
  </wsdl:types>
  <wsdl:message name="DrawRequest" sawsdl:modelReference="...#operation" >
      <wsdl:part name="Request" element="tns:DrawMsg" sawsdl:modelReference="...#operation" />
  </wsdl:message>
  <wsdl:message name="DrawResponse">
      <wsdl:part name="Response" element="tns:String" />
  </wsdl:message>...
  <wsdl:message name="EquipmentChangeState">
      <wsdl:part name="EquipmentChangeState" element="tns:EquipmentChangeState"
    sawsdl:modelReference="...#currentState ...#previousState ...#operation" />
  </wsdl:message>
  <wsdl:portType name="RobotPortType" ... sawsdl:modelReference="...#RobotVariableBinding2">
      <wsdl:operation name="Draw">
        <sawsdl:attrExtensions sawsdl:modelReference="...#robotOperationRule" />
          <wsdl:input name="DrawRequest" message="tns:DrawRequest" />
          <wsdl:output name="DrawResponse" message="tns:DrawResponse" />
      </wsdl:operation>
      <wsdl:operation name="EquipmentChangeState">
        <sawsdl:attrExtensions sawsdl:modelReference="...#operationCompletedRule ...#
            operationStartedRule" />
          <wsdl:output name="EquipmentChangeState" message="tns:EquipmentChangeState" />
      </wsdl:operation>
  </wsdl:portType>
  <!-- This embedded OWL document imports the actual OWL model that contains
  the concepts referenced in the SAWSDL annotations. -->
<rdf:RDF ...>
  <owl:Ontology>
    <owl:imports rdf:resource="http://www.pe.tut.fi/fast/owl/models/Fastory-abox.owl" />
  </owl:Ontology>
  <process:Binding
    rdf:about="...#RobotVariableBinding2">
    <process:toVar
      rdf:resource="...#robot" />
    <process:valueObject
      rdf:resource="...#robot2" />
  </process:Binding>
</rdf:RDF>
</wsdl:definitions>
```

Listing 5.9: The WSDL operations of the robot service contain several SAWSDL model references to SWRL rules.

Table 5.6: The Robot service WSDL operations are linked to SWRL rules through SAWSDL annotations.

Operation Name	**SAWSDL Model Reference**
Draw (not robot 1)	robotOperationRule
replacePaper (only robot 1)	palletLoadRule, palletUnloadRule
EquipmentChangeState (robot 1)	assemblyLoadedRule, assemblyUnloadedRule
EquipmentChangeState (not robot 1)	operationStartedRule, operationCompletedRule

is therefore absent from the listing, includes model references to two SWRL rules: *palletLoadRule* and *palletUnloadRule*. The former rule applies to the case that the pallet is initially blank and the latter to the case that the pallet already carries an assembly, which is then replaced. Therefore, Service Monitor generates two *results* for the corresponding OWL-S *process*. The *results* represent the two alternative consequences of the *replacePaper* operation, namely the placement and the replacement of a paper. The WSDL operation *Draw* on line 46 in Listing 5.9 only contains a model reference to the SWRL rule *robotOperationRule*, and hence Service Monitor only generates one *result* for the *process*, which corresponds to the event that the selected operation is performed on the assembly carried by the pallet occupying the robot processing location. Table 5.6 shows the SAWSDL model references attached to the robot service operations, and Table 5.7 lists the three SWRL rules referred to by the invokable operations. The WSDL notification operations refer to four somewhat different SWRL rules, from which Service Monitor extracts domain model update rules.

The generated OWL-S *results* include condition and effect expressions, which Service Monitor derives from the corresponding SWRL rules. However, Service Monitor applies the variable value bindings defined in the OWL model embedded in the service WSDL file. For example, the OWL model in Listing 5.9 contains one variable binding, *RobotVariableBinding2*, on lines 61-67. It specifies that the SWRL variable *robot* should be substituted with the OWL individual *robot2* in the condition and effect expressions derived. Because the WSDL port type containing the aforementioned operations includes an SAWSDL model reference to the *binding* on line 44, the *binding* will be applied in deriving the condition and effect expressions for the *results* in all of the OWL-S *processes*.

The conditions for the generated *results* are derived by copying all atoms from the rule antecedents and applying the specified variable substitutions. For example, *RobotVariableBinding2* would be applied by substituting each occurrence of the variable *robot* in Table 5.7 with the OWL individual *robot2*. In addition, SWRL

Table 5.7: The Fastory domain ontology defines three SWRL rules representing the tasks carried out by robots.

Rule Name	**Rule Expression**
palletLoadRule	$hasProcessingLocation(?robot, ?zone) \wedge$ $hasInputStorage(?robot, ?storage) \wedge$ $stores(?storage, ?assembly) \wedge$ $isOnTopOf(?assembly, ?newTop) \wedge$ $isTopOf(?assembly, ?storage) \wedge$ $hasPallet(?zone, ?pallet) \wedge$ $canPerformOperation(?robot, ?operation) \wedge$ $PalletMoveOperation(?operation) \wedge swrlb :$ $noValue(?pallet, carriesProduct) \Rightarrow swrlb :$ $remove(2) \wedge swrlb : remove(3) \wedge swrlb :$ $remove(4) \wedge carriesProduct(?pallet, ?assembly) \wedge$ $isTopOf(?newTop, ?storage)$
palletUnloadRule	$hasProcessingLocation(?robot, ?zone) \wedge$ $carriesProduct(?pallet, ?assembly) \wedge$ $hasPallet(?zone, ?pallet) \wedge$ $canPerformOperation(?robot, ?operation) \wedge$ $hasInputStorage(?robot, ?inputStack) \wedge$ $isTopOf(?oldInputTop, ?inputStack) \wedge$ $isOnTopOf(?oldInputTop, ?newInputTop) \wedge$ $stores(?inputStack, ?oldInputTop) \wedge$ $hasOutputStorage(?robot, ?storage) \wedge$ $isTopOf(?oldTop, ?storage) \wedge$ $PalletMoveOperation(?operation) \Rightarrow swrlb :$ $remove(1) \wedge swrlb : remove(5) \wedge swrlb :$ $remove(6) \wedge swrlb : remove(7) \wedge swrlb :$ $remove(9) \wedge stores(?storage, ?assembly) \wedge$ $isTopOf(?assembly, ?storage) \wedge$ $isOnTopOf(?assembly, ?oldTop) \wedge$ $carriesProduct(?pallet, ?oldInputTop) \wedge$ $isTopOf(?newInputTop, ?inputStack)$
robotOperationRule	$hasRobot(?cell, ?robot) \wedge hasRobotStatus(?robot, blocked) \wedge$ $hasProcessingLocation(?robot, ?zone) \wedge$ $hasPallet(?zone, ?pallet) \wedge hasRfidTag(?pallet, ?rfid) \wedge$ $carriesProduct(?pallet, ?assembly) \wedge$ $canPerformOperation(?robot, ?operation) \wedge$ $insertsComponent(?operation, ?component) \wedge swrlb :$ $noValue(?assembly, hasComponent, ?component) \Rightarrow$ $hasComponent(?assembly, ?component)$

variables are substituted with the generated OWL-S process input or output parameters if the corresponding WSDL message parts contain model references to the

variables. For example, on line 35 in Listing 5.9, the input message part of the *Draw* operation contains a model reference to the SWRL variable *operation*. Therefore, the variable is substituted with the generated OWL-S input parameter when copying the conditions from the SWRL rule *robotOperationRule* to the generated OWL-S *result*. The effects are derived by applying the same variable substitutions to the atoms in the SWRL rule consequents.

The WSDL files of the other robot services contain similar SAWSDL annotations referring to the same SWRL rules. However, the embedded OWL documents contain different variable substitutions. For example, the WSDL file of the service corresponding to robot 3 contains an OWL document that substitutes the SWRL variable *robot* with the OWL individual *robot3*. The parts that need to be altered for the different instances of the robot service appear in bold font in Listing 5.9.

Listing 5.10 shows the abbreviated XML code for the OWL-S *process* generated from the robot 2 *Draw* operation. For example, Service Monitor generates the XSLT script for the XML schema element *DrawMsg* defined on line 25 in Listing 5.9 and inserts the script on line 43 in Listing 5.10. However, the XSLT script, as well as the SWRL condition expression of the process result, are quite verbose and therefore omitted in the listing. The SWRL effect expression of the process result is less verbose than the condition, since it only specifies that executing the process will cause a new component to be attached the assembly carried by the pallet at the robot 2 processing location. The effect is defined on lines 5-14 in Listing 5.10.

In the example presented in this section, Service Monitor generated condition and effect expressions in SWRL. Alternatively, Service Monitor could be configured to use SPARQL as the expression language, which might result in more human-readable derivation results. When Service Monitor is configured to use SPARQL, the OWL-S derivation process is otherwise identical except that the condition and effect expressions in the generated OWL-S processes are different.

5.7 Application Example of SWRL-based Semantic Web Service Composition

This section exemplifies the semantic web service composition approach presented in Section 3.6. Because the experiments involve the application domain presented in Section 5.1.4, the domain model illustrated in Figure 5.24 is reused in this application scenario.

To initialize the service composition framework, the Ontology Service and Service Monitor services must be deployed, and the domain model on Ontology Service must be initialized by invoking the *SetBaseOntology* method.

The objective for the service composition scenario is that the storage reserved for

```
<process:AtomicProcess rdf:about="...#DrawProcess">
  <process:hasResult>
    <process:Result rdf:about="...#DrawProcessResult1">
      <process:hasEffect>
        <expression:SWRL−Expression>
          <expression:expressionObject rdf:parseType="Collection">
            <j.3:IndividualPropertyAtom>
              <j.3:argument2 rdf:resource="...#component"/>
              <j.3:argument1 rdf:resource="...#assembly"/>
              <j.3:propertyPredicate rdf:resource ="...#hasComponent"/>
            </j.3:IndividualPropertyAtom>
          </expression:expressionObject>
          <expression:expressionLanguage rdf:resource="...#SWRL"/>
        </expression:SWRL−Expression>
      </process:hasEffect>
      <process:inCondition>
        <expression:SWRL−Condition>...</expression:SWRL−Condition>
      </process:inCondition>
    </process:Result>
  </process:hasResult>
  <process:hasOutput>
    <process:Output rdf:about="...#DrawOutput1">
      <process:parameterType rdf:datatype="...#anyURI"
      >http://www.w3.org/2002/07/owl#Thing</process:parameterType>
    </process:Output>
  </process:hasOutput>
  <process:hasInput rdf:resource="...#DrawInput1"/>
  <service:describes>
    <service:Service rdf:about="...#DrawService">
      <service:supports>
        <grounding:WsdlGrounding rdf:about="...#DrawGrounding">
          <grounding:hasAtomicProcessGrounding>
            <grounding:WsdlAtomicProcessGrounding rdf:about="...#DrawAtomicGrounding">
              <grounding:wsdlOutput>
                <grounding:WsdlOutputMessageMap>
                  <grounding:owlsParameter rdf:resource="...#DrawOutput1"/>
                  <grounding:xsltTransformationString>...</grounding:xsltTransformationString>
                </grounding:WsdlOutputMessageMap>
              </grounding:wsdlOutput>
              <grounding:wsdlInput>
                <grounding:WsdlInputMessageMap>
                  <grounding:owlsParameter rdf:resource="...#DrawInput1"/>
                  <grounding:xsltTransformationString>... </grounding:xsltTransformationString>
                  <grounding:wsdlMessagePart rdf:datatype="...#anyURI">.../FastoryRobot2.wsdl#Request
                      </grounding:wsdlMessagePart>
                </grounding:WsdlInputMessageMap>
              </grounding:wsdlInput>
              <grounding:wsdlDocument rdf:datatype="...#anyURI">.../FastoryRobot2.wsdl</
                  grounding:wsdlDocument>
              <grounding:wsdlOperation>
                <grounding:WsdlOperationRef>
                  <grounding:operation rdf:datatype="...#anyURI">.../FastoryRobot2.wsdl#Draw</
                      grounding:operation>
                </grounding:WsdlOperationRef>
              </grounding:wsdlOperation>
              <grounding:owlsProcess rdf:resource="...#DrawProcess"/>
            </grounding:WsdlAtomicProcessGrounding>
          </grounding:hasAtomicProcessGrounding>
        </grounding:WsdlGrounding>
      </service:supports>
      <service:describedBy rdf:resource="...#DrawProcess"/>
    </service:Service>
  </service:describes>
</process:AtomicProcess>
```

Listing 5.10: The OWL-S Process generated from the *Draw* WSDL operation includes one OWL-S Result.

completed products would contain a product comprising a type 1 phone frame, a type 3 keypad, and a type 3 screen. Listing 5.11 contains the goal expression sent to Service Monitor in this case. The objective can be submitted to Service Monitor either by using the GUI displayed on the host computer or by invoking the *StartGoal* operation in the Service Monitor web service interface.

Subsequent to receiving the objective, Service Monitor creates a new goal process aiming to achieve it. Each goal process commences with a planning phase, which is succeeded by a plan execution phase. If either phase fails, the goal process automatically recommences and considers the current domain state as the initial state of the planning problem.

Before starting to solve the problem, the planner performs an initialization phase in which it instantiates the goal expression by resolving variables occurring in the goal expression. The domain model ABox contains five assemblies and two storages, one storage for blank product templates and one for completed products. Thus, there are 10 possible value combinations for the variables. Furthermore, each of the resulting statements is feasible in that it is present in the positive effects of at least one action in the planning problem. Hence, after the planner has instantiated the goal, the remaining goal is a list of ten conjunctions, each of which contains four statements containing no variables but only ground instances.

The planning initialization phase takes a few seconds, and the actual planning phase takes approximately 30 seconds. Subsequent to the planning phase, Service Monitor begins executing the obtained solution plan.

The duration of the plan execution phase is longer when the approach is applied on the actual production system instead of the web services implemented using Java, because the actual assembly operations take longer to complete than the simulated time delays in the Java-based services.

Regardless of whether the approach is applied on the actual or the simulated system, the plan execution phase includes additional delays caused by the latency in receiving event notifications from the web services, applying the update rules, as well as verifying OWL-S process conditions and effects. The delays occur for each

```
1  PREFIX generic:<http://www.pe.tut.fi/fast/ontologies/ProductionSystem.owl#>
2  PREFIX fastory:<http://www.pe.tut.fi/fast/ontologies/Fastory.owl#>
3  PREFIX abox:<http://www.pe.tut.fi/fast/owl/models/Fastory-abox.owl#>
4  ASK {
5  ?assembly generic:hasComponent abox:phoneFrame_1 .
6  ?assembly generic:hasComponent abox:phoneKeypad_3 .
7  ?assembly generic:hasComponent abox:phoneScreen_3 .
8  ?storage  fastory:stores  ?assembly .
9  }
```

Listing 5.11: The production goal requires attaching three components to a phone assembly and transporting the finished product into storage.

executed OWL-S atomic process and are approximately a few hundred milliseconds per process.

5.8 Application of the Cloud Resource Utilization Approach

The Cloud Gateway service presented in Section 3.7 has been tested both on a private computing cloud created using the Eucalyptus [66] software framework and on the Amazon EC2. This section will first present the experiment setup and then describe the test results.

5.8.1 The Experiment Setup

The private cloud consists of only one computing cluster composed of a single desktop running a Linux operating system and Eucalyptus version 1.6.1. The restricted computing cloud limits, for example, the number of virtual machines that may be created; the performed experiments involve a maximum of two parallel virtual machine instances. Nonetheless, even such a limited setting suffices for testing the proposed approach.

The virtual machines in the computing cloud are created from a disk image that readily includes all of the software components necessary for starting the Cloud Gateway server application. The Euca2ools[3] command line utilities allow the upload of the image to the cloud and the creation of virtual machines based on the image.

The Service Explorer application described in Section 4.1 facilitates interaction with the Cloud Gateway services. The application includes a graphical user interface allowing, for example, inspection of web services and invocation of their operations. In the experiments described in this section, Service Explorer is run on a laptop connected to the same local network as the desktop hosting the private computing cloud. Thus, Service Explorer is able to automatically detect the web services started on the virtual machines.

Each virtual machine executes a separate copy of the Cloud Gateway server application. Figure 5.27 illustrates the experiment topology including two virtual machine instances in the private cloud.

5.8.2 Performance Measurement

A test application called Cloud Gateway Performance Test allows measurement of the delays present in the Cloud Gateway service operation. The test application replaces Service Explorer in Figure 5.27. However, the simple user interface provided by the test application is specifically designed for interaction with the Cloud

[3]Available at https://www.eucalyptus.com/download/euca2ools. Accessed on 2014-06-16.

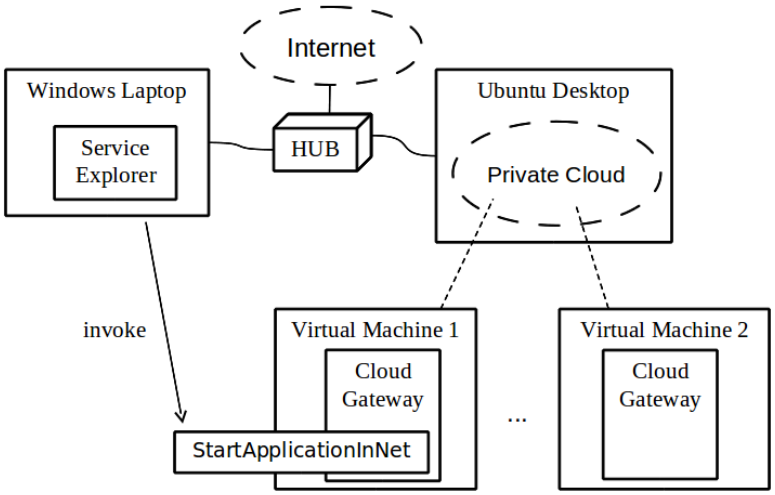


Figure 5.27: The test arrangement includes two physical machines, one of which hosts a private cloud containing virtual machines (VMs).

Gateway service. The purpose of the test application is to measure the time required for deploying several independent web services in a computing cloud. The test application first invokes the *AddApplication* method to register an application in the Cloud Gateway application library and then sequentially invokes the *StartApplicationInNet* operation to execute the application a number of times specified by the user. After each *StartApplicationInNet* request, the test application waits for the Cloud Gateway to send a *ServiceStarted* notification before sending the next request. The user interface includes text fields for specifying the JAR file URL and the number of times to execute the JAR with Cloud Gateway. In addition, the performance test application allows the user to specify threshold values, which the application then requests Cloud Gateway to use by invoking the *SetThresholdInNet* operation. The test results indicate the delay before a WS-Discovery Hello message was received from each of the services deployed. Figure 5.28 shows a screenshot of the Performance Test application.

Experimenting with different threshold values reveals that if only a RAM usage threshold were used, it should be set to at most 0.98 because the operating system never appears to let the RAM utilization reach 99 percent but retains a small amount of memory as work space and compensates the missing memory with swap file usage. For example, with one gigabyte of RAM, the memory utilization typically reaches 98 percent after Cloud Gateway has started 28 conveyor service server applications, after which the proportion of used RAM fluctuates only marginally. However, the increased page file usage burdens the CPU, resulting in very poor performance. To prevent the CPU load from excessively increasing, a CPU utilization threshold value should be specified.

In one of the test runs, setting the RAM threshold to 0.99 and CPU utilization

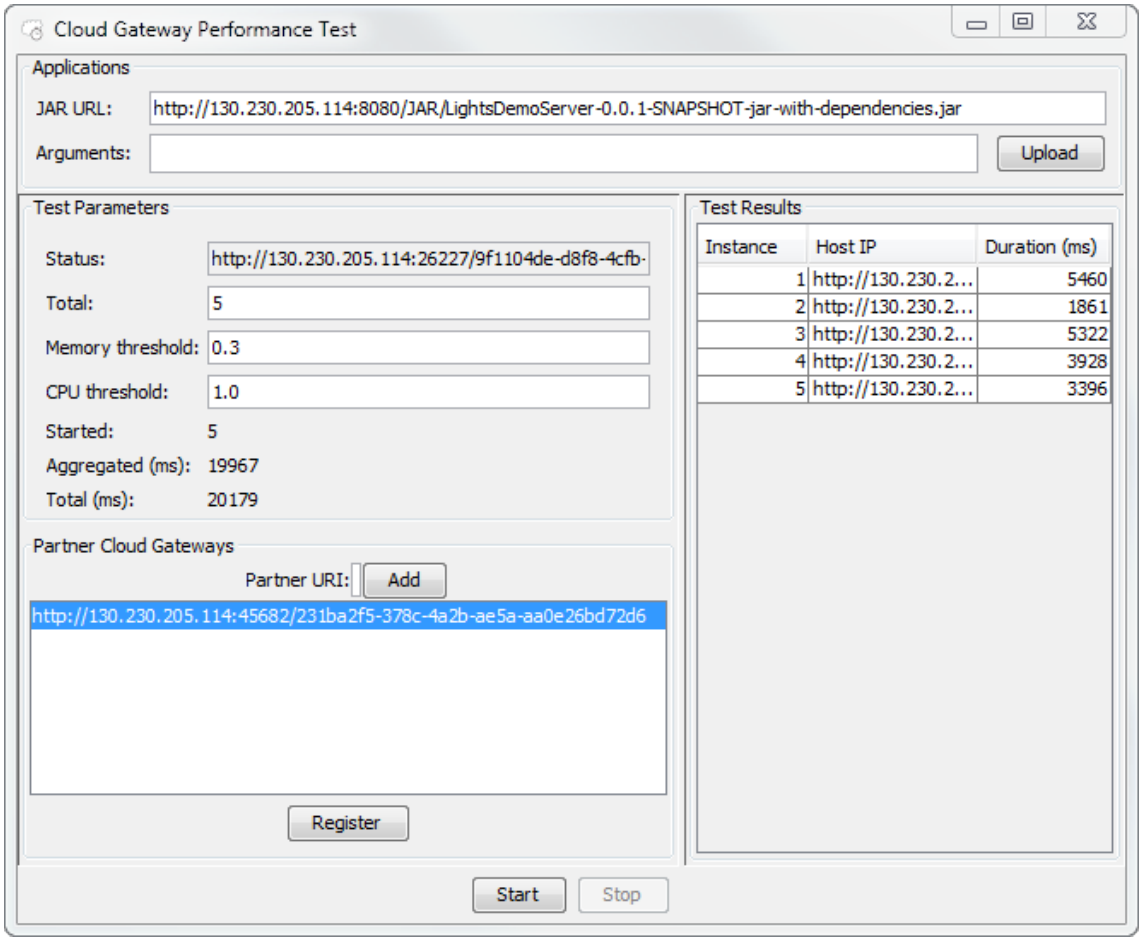


Figure 5.28: The Cloud Gateway client application measures service deployment durations using Cloud Gateway services.

threshold to 6.0 resulted in 42 applications being started before the CPU threshold was exceeded. Cloud Gateway started the applications in 217 seconds. However, the maximum number of applications and the start-up delay vary between different test runs. Given that the virtual machine hosting Cloud Gateway comprises only one (virtual) CPU, the load factor of 6.0 indicates that, on average, only five processes are queuing for CPU time. However, the web services, including Cloud Gateway, running on the machine seemed unable to respond to requests within the communication time out durations. Logging in to the virtual machine revealed that the load average had exceeded 60. Afterwards, the virtual machine became unreachable. Apparently, since the load average is computed over the previous minute [115], it is difficult to use it as a measure of the workload of the machine at a specific instant. On the other hand, the applications may temporarily have to queue for processing time at start-up, while later they will require less CPU usage.

On a virtual machine with only 256 megabytes of RAM, the memory utilization exceeds 98 percent after Cloud Gateway has only started six server applications, and the overall delay is 22 seconds. When the memory threshold is set to 99 percent

Table 5.8: The number of conveyor service applications that can be started on a virtual machine with 1.7 GB of RAM.

Memory threshold	**Reason for termination**	**Number of instances**	**Total duration (s)**
0.9	memory	28	138
0.98	memory	33	173
1	CPU $>$ 50	85	652
1	failure	93	1379

and CPU threshold is set to 6.0, Cloud Gateway starts nine server applications, but finally the virtual machine becomes unreachable.

The application start-up delay begins to increase rapidly after Cloud Gateway has started a certain number of applications. This is obviously caused by the virtual machine having to compensate the lack of physical memory with page file usage. Moreover, the responsiveness of the applications running on a virtual machine is very poor when the machine is simultaneously executing several applications.

In addition, experiments have been conducted running the Cloud Gateway service on remote virtual machines leased from the Amazon EC2 cloud. In the experiments, each virtual machine hosting a Cloud Gateway service has been allocated 1.7 gigabytes of RAM. However, Table 5.8 shows the test results using a single virtual machine in the EC2 cloud. The table shows the memory threshold, number of started application instances and overall start-up durations. It also lists the reasons why Cloud Gateway ceased to start new server applications.

The last row in Table 5.8 represents a test scenario where the CPU threshold was set to 100. In this case, the client connection to the virtual machine abruptly terminated while starting the 94th application instance, apparently due to the excessive workload on the virtual machine.

5.8.3 Performance Measurement in a Network Setting

This subsection summarizes performance tests carried out in a setting of two Cloud Gateways running on separate virtual machines in a private computing cloud. Each of the virtual machines is allocated one gigabyte of RAM and five gigabytes of disk space. While the Cloud Gateway Performance Test application directly communicates only with the main instance, it invokes the *RegisterCloudGateway* operation on the main instance to add the auxiliary instance to the Cloud Gateway network. The memory and CPU thresholds set in the user interface are submitted to each Cloud Gateway in the network.

In the scenario of two Cloud Gateway instances, the main instance will serve the

first application requests. However, once the main instance exceeds the memory usage threshold, it commences delegation of incoming requests to start new applications to the auxiliary instance.

For example, in one of the test runs, the memory threshold of the two Cloud Gateways was set to 98 percent, while the CPU threshold was set to five. Finally, the performance test application started requesting the main Cloud Gateway instance to start instances of the conveyor service server application. The main instance exceeded the memory threshold after starting the 28th server application and started delegating the requests to the auxiliary instance on the other virtual machine. The auxiliary instance was able to start 27 applications before exceeding the memory threshold. Hence, a total of 55 application instances were started, and the total duration was approximately 210 seconds.

5.8.4 Inter-Cloud Experiment Scenario

To experiment web service orchestration across different computing clouds, experiments can be performed using two remote virtual machines and one local virtual machine. Unlike the previous experiment scenarios, the experiments described in this subsection are performed without the Cloud Gateway Performance Test application. The remote virtual machines are leased from the Amazon EC2 cloud, while the local virtual machine is running in a private computing cloud.

Each virtual machine hosts one Orchestration Engine web service and three virtual conveyor web services. In addition, a monitor application communicates with the Orchestration Engine service running on the local virtual machine. Figure 5.29 illustrates the experiment topology and the web service invocations during the experiments.

The experiment consists of a cycle commencing when the Orchestration Engine on virtual machine 1 is requested to execute a BPEL process orchestrating the three conveyor web services. At the end of the process, the Orchestration Engine on virtual machine 1 requests the Orchestration Engine on virtual machine 2 to execute a similar process, which is represented by step 4 in Figure 5.29. Then, the Orchestration Engine on virtual machine 3 executes a similar BPEL process, which finally requests the Orchestration Engine on virtual machine 1 to again execute the BPEL process (step 12). Hence, the cycle continues indefinitely, so that only one Orchestration Engine is executing a BPEL process at a time.

To measure cycle durations, a client application monitors the Orchestration Engine service on the local virtual machine. Whenever the Orchestration Engine sends a notification signaling that it has commenced execution of a BPEL process, the client application records the duration of the elapsed interval since the previous notification. In addition, the monitor application determines the minimum, maximum

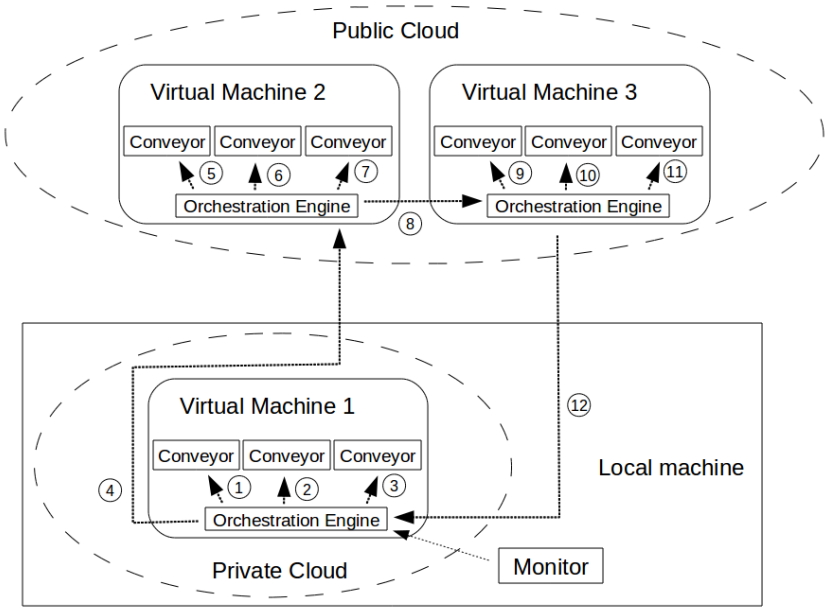


Figure 5.29: The private cloud is hosted on a local machine. Each virtual machine hosts four web services.

and average interval duration.

While research on BPEL-based web service orchestration is presented in [81], the Orchestration Engine web service is described in [80] and [49].

Table 5.9 contains the minimum, maximum and average intervals observed in an experiment consisting of 20 cycles. To provide a reference point, the table includes the results for a repetition of the experiment performed so that all of the web services were running on the local machine. The results in Table 5.9 indicate that the average cycle duration is approximately two seconds longer when using computing clouds. This constitutes less than five percent of the average cycle duration. The minor performance degradation is presumably caused by the network traffic between the web services on different virtual machines. Nonetheless, network traffic is unavoidable when the Orchestration Engine services reside on different machines.

Table 5.9: The duration of 20 cycles is quite similar regardless of whether cloud resources are used instead of local resources.

	Minimum (ms)	**Maximum (ms)**	**Average (ms)**
1 local VM, 2 remote VMs	47471	49106	47961
Only local web services	45519	46787	45682

5.9 Summary of Application Examples

The application examples presented in this chapter commenced with applying the traditional web service standards and orchestration tools in the factory automation domain. The subsequent examples involved exploiting semantic web service descriptions, which required the use of additional tools. In the presented examples, the additional tools were mainly implemented as semantic web services that co-operated to form a framework for the orchestration and composition of semantic domain web services. In particular, the methods presented in Chapter 3 were employed to develop semantic web service descriptions with reduced effort and to apply semantic descriptions in maintaining an accurate semantic model of a production system. Finally, Section 5.8 experimented with deploying web services on cloud resources, albeit with no particular emphasis on semantic technologies.

The presented application examples have employed different technologies related to semantic web services, and Table 5.10 summarizes the relationships between individual application examples and technologies. On the one hand, some of the technologies, such as WSDL, are de facto standards and applied in each of the examples. On the other hand, both the approaches presented in Chapter 3 and the application examples presented in this chapter have omitted some considerable technology alternatives, such as WSMO.

While Table 5.10 indicates the technologies applied in each application example, it provides no absolute indication of their applicability to each of the exemplified approaches. For example, the OWL-S derivation approach presented in Section 3.5 can generate SPARQL expressions, but Section 5.6 exemplified only the derivation of SWRL expressions.

While the presented experiments show that the methods outlined in Chapter 3 can be applied in practice, the methods evidently introduce additional workload.

Table 5.10: The application examples involve different web-service-related standards.

Technology/Section	**5.2**	**5.3**	**5.4**	**5.5**	**5.6**	**5.7**	**5.8**
WSDL	✓	✓	✓	✓	✓	✓	✓
BPEL	✓	✓	✓				✓
SAWSDL		✓	✓	✓	✓	✓	
OWL		✓	✓	✓	✓	✓	
SPARQL and SPARQL/Update		✓	✓	✓		✓	
SWRL				✓	✓	✓	
OWL-S		✓	✓	✓	✓	✓	
WSMO							

For example, the initialization of the semantic web service composition framework requires additional effort. The deficiencies and potential remedies will be described in more detail in Section 6.2.

6. CONCLUSIONS

This dissertation has presented new methods of controlling SOA-based production systems. While the new methods can drastically reduce the effort required to prescribe workflows, they require that effort is invested in devising semantic service descriptions. However, service descriptions must be developed only once, and the methods eliminate the need to manually prescribe new process workflows whenever a change occurs in the production system or the pursued production goal.

6.1 Contributions

This section summarizes the contributions this dissertation makes to SOA-based production system control.

6.1.1 Experience on BPEL-based Orchestration

This dissertation has exemplified the issues that can arise in service orchestration in the domain of factory automation. BPEL can be considered the de facto standard language for the orchestration of web services. While a single step in the highest-level business process expressed in BPEL can take several weeks, it is possible to conceptually arrive to factory floor by clarifying each service with the corresponding BPEL sub-process, eventually considering the level of atomic processes, such as the activation of a motor or the examination of a sensor status.

Although BPEL is supported by a number of tools, this dissertation has suggested that none of the tools perfectly addresses the needs of service orchestration, nor provides full flexibility of the standards. One possible reason for the deficiencies is that some of the standards still have a 'candidate recommendation' status of W3C (e.g. [118]) or have been accepted as standards quite recently.

In addition, this dissertation has presented new tools for the execution of BPEL processes. Instead of solving the problems identified in the prevalent tools, the new tools apply an approach to executing BPEL processes that is fundamentally different to the standard solution, in which BPEL processes are deployed as web services on a server. On the one hand, executing BPEL processes as scripts somewhat restricts the range of supported BPEL processes and constructs. On the other hand, when applicable, the approach removes the need of a server engine and is somewhat more efficient than deploying each BPEL process on a server.

6.1.2 A Semantic Web Service Orchestration Framework

This dissertation has proposed an approach to applying OWL in dynamically maintaining a description of the states of web services, while OWL-S is applied in determining the suitable partner web services to use in carrying out orchestration instructions. Invoking operations on a web service causes changes in its internal state, which can be detected by listening to notifications sent by the service. Dedicated system components are responsible for reasoning on the current state of the system and deciding on the following actions. The proposed approach and the performed experiments are initial steps towards common utilization of semantic web services in the factory floor.

Semantic web services can considerably facilitate the control of production equipment. Because the overall process instructions can be executed by a web service, the main controller of a production system can basically be located anywhere regardless of the location of the actual production system. Thus, it can be effortlessly replaced, and it is unexposed to the hazards of a production environment. Indeed, dedicated server equipment is unnecessary, as the controller service can be deployed, for example, on a virtual machine in a computing cloud. New virtual machines can be started as production requests arrive, and once the products are finished, the virtual machines may be terminated, and the computing resources can be allocated to other activities.

6.1.3 A Semantic Web Service Composition Framework

This dissertation has described a set of specialized web services that jointly apply OWL and SPARQL in composing and invoking semantic domain web services. The results show that the approach allows automatic achievement of complex goals while maintaining a temporally accurate domain model, provided that the web services are augmented with adequate semantic descriptions and event notifications.

While OWL, SPARQL, and SPARQL/Update are highly expressive languages, developing an efficient planning algorithm appears to require that the semantic descriptions are transformed into more suitable formats, such as PDDL, prior to planning. The approach would allow domain engineers to model the systems and web services in the abovementioned expressive and widely supported languages, while still enabling the efficient and automatic composition of web services.

The reuse of previously composed OWL-S processes requires that Service Monitor stores each process composed into the service model, which may excessively increase the number of statements in the model and the knowledge access delays. Moreover, modifications in the set of semantic web services available may render previously composed *composite* OWL-S processes suboptimal or inapplicable. Nonetheless, it

will be unnecessary to store and reuse previously composed *composite* processes after the development of a more sophisticated planning algorithm.

6.1.4 Domain Model Update Methods

This dissertation has presented two event-based approaches to updating production system models and highlighted their main differences. In both approaches, a web service called Ontology Service hosts an OWL domain model, and another service, either Ontology Manager or Service Monitor, sends update requests to Ontology Service based on event notifications. Both approaches have shown negligible domain model update delays in the test scenarios involving simulated production systems. However, the implementation still requires some optimization, and a more comprehensive performance evaluation in more demanding test scenarios should be carried out before industrial application.

On the one hand, the Ontology Manager based approach fails to fully achieve the benefits of semantic web services, since it ignores semantic web service descriptions. Moreover, developing the Ontology Manager update rules and particularly the embedded XSLT scripts may be difficult. On the other hand, semantic web service descriptions typically refer to a specific domain model, and the update rules make it possible to use whichever domain model is the most appropriate for the application scenario, regardless of the semantic web service descriptions. In addition, the Ontology Manager GUI could be improved so that it automatically generated the XSLT scripts.

The Service Monitor based approach fully considers semantic service descriptions but can only be applied once the domain web services have been enriched with such descriptions. Fortunately, Service Monitor is able to automatically generate the semantic descriptions, provided that the service WSDL files are appropriately annotated to refer to an accessible domain model. Thus, neither one of the two approaches is clearly superior to another. Instead, the applicability of each approach depends on the scenario.

6.1.5 Automated OWL-S Generation

There are no standard rules for formulating semantic descriptions or attaching them to web services. Nonetheless, to facilitate automatic web service composition and execution, the semantic descriptions must be somewhat detailed. Consequently, formulating dedicated semantic descriptions for all of the individual services can be tedious and error-prone in application scenarios involving dozens of services. This dissertation has proposed a set of conventions on the application of SAWSDL annotations that facilitate the automatic derivation of detailed OWL-S descriptions from

service WSDL documents. The approach is particularly useful in scenarios involving several services of the same type and reduces the effort of creating numerous semantic service descriptions to only defining a modest set of SWRL rules and annotating the service WSDL files accordingly.

The approach presented in this dissertation automates the generation of executable OWL-S processes that include condition and effect expressions formulated in either SWRL or SPARQL. On the one hand, applying the approach requires quite extensive use of SAWSDL annotations. On the other hand, simply annotating the WSDL files is considerably less laborious and difficult than manually duplicating OWL-S descriptions for several instances of a web service, since the descriptions are typically somewhat verbose and contain several complex expressions, such as XSLT scripts.

6.1.6 More Optimal Cloud Resource Utilization

This dissertation has proposed a web service that facilitates the use of computing cloud resources. The approach allows the use of cloud resources without knowledge on the cloud interface or internal composition. In particular, the application examples show that a client can use the cloud resources by invoking simple web service operations, without directly interacting with the leased virtual machines.

The performance of the proposed approach has been experimentally evaluated. The experiments have shown that, while the automated execution of applications is effortless, the resource limits of the underlying virtual machine are eventually reached as the number of executed applications increases. Moreover, exhausting the resources over a certain point tends to considerably decrease application responsiveness. Nonetheless, the experiments have also shown that networks comprising several Cloud Gateway services are able to automatically balance the workload between different virtual machines.

6.2 Potential Enhancements

This section identifies possible enhancements to the proposed approaches and the implemented software tools.

6.2.1 Enhanced BPEL Support

The Orchestration Tools framework currently provides only limited BPEL support and flexibility. In particular, several BPEL constructs remain unsupported, and the tools are capable of only synchronous BPEL execution. Asynchronous execution would probably require a web-server-based approach.

6.2.2 More Advanced use of Semantic Service Descriptions

Further research is required to achieve automated service selection based on entire semantic service descriptions. While Service Monitor currently considers only the input and output types of OWL-S Processes in semantic service selection, it should also consider the descriptions of the preconditions and results of the web service operations. Since preconditions and results are essentially represented as expressions, comparing their semantic meanings is problematic.

The selection of web services for BPEL partner links in the Orchestration Engine component should be further developed. Because only semantic information is relevant in expressing service requirements, an apparently compatible service may have an interface that is syntactically incompatible with the current BPEL process. Hence, Orchestration Engine must modify the executed BPEL process so that it refers to the new WSDL port types and operations after determining the services that are semantically compatible with the partner links. However, the logic for modifying the orchestration instructions to apply to unexpected service interfaces remains largely unimplemented in Orchestration Engine. Modifying the various expressions that can be embedded into a BPEL process and refer to syntactic elements appears to be a particularly difficult problem. On the other hand, Service Monitor could probably be extended to provide support in BPEL modification as well.

6.2.3 Support for Alternative Formalisms

While orchestration instructions for the Orchestration Engine are currently devised in BPEL, other languages should also be considered. In particular, OWL-S might be a competitive alternative for BPEL, since it is able to express complex processes and naturally uses semantic data, while BPEL purely relies on syntactic web service interfaces.

The current Orchestration Engine implementation uses the syntactic WSDL description when invoking a web service. However, while the OWL-S specification mainly presents WSDL groundings, it supports other types of groundings as well. Hence, web service invocation based on the semantic OWL-S descriptions might allow the invocation of web services without WSDL descriptions. However, invoking web services by executing their OWL-S processes would require either extending the BPEL specification similarly to the BPEL4SWS [65] specification or entirely switching to another modeling language, such as OWL-S, in specifying workflows.

6.2.4 Support for Real-time Requirements

Because the proposed web service orchestration framework is implemented in Java, and the communication between the individual components depends on network con-

nections, the operation delays are indeterministic. For example, the communication between Ontology Service and Ontology Manager includes the risk that a query is performed while an update operation has been requested but is still pending. Moreover, an event notification sent by a domain web service may be delayed because of high network traffic, causing Orchestration Engine to make decisions based on outdated information. On the other hand, the semantic service composition process, in particular, includes noticeable delays, which may be unacceptable in settings imposing strict real-time requirements. While some optimization can be performed to the search algorithm, a domain-independent planning process must generally explore a vast state space. Thus, high real-time performance is difficult to achieve.

Real-time or deterministic aspects are largely ignored in this dissertation, since the main focus is on the overall approach. At the implementation level, the reasoning libraries may require optimization to allow deterministic performance in some specific cases. For example, the libraries might perform unnecessary memory reservation and release, since they are designed to be integrated in desktop environment applications in which delays of a few seconds are tolerable. Nonetheless, the problem is more related to implementation details than the actual research aspects.

The main research problems include estimation of the performance needs of reasoning to provide results before commencing the execution of a workflow. For example, the requirement for deterministic performance can be mitigated if time-consuming and indeterministic operations are performed only at initialization phase before commencing the actual execution of a web service workflow. In the case of geographically distributed applications, the communication delays can be estimated at the system design phase and be taken into account at the right level of control implementation. For example, considering the control of a robot manipulator, the trajectories can be calculated remotely and uploaded to the manipulator controller with arbitrary communication delays. Nevertheless, the actual interpolated motion of the manipulator joints can be locally performed, complying to strict real-time requirements. Some research that is related to distributed applications as well as considers real-time performance and avoiding unnecessary memory reservations in Java-based controllers can be found in [59; 50; 15].

6.2.5 Increased Decentralization

In the web service composition pattern proposed, decision-making is solely the responsibility of the Service Monitor web service. Currently, exactly one instance of Service Monitor is assumed to exist at a time. While Service Monitor can compose and execute solution plans that involve concurrent operations, parallel goal processes typically lead to a conflict at plan execution time. Moreover, the case of several Service Monitor instances would require the development of a coordination

mechanism.

All of the presented approaches are vulnerable to network disruptions. In particular, an event notification lost during transmission may render the domain model inconsistent with the actual system status. To protect against such errors, the approaches can incorporate periodic polling of web service states. Nonetheless, decentralization could be increased by developing a collaboration mechanism between several Ontology Service instances.

Semantic integration between potential alternative ontologies is mainly ignored in this dissertation. Consequently, the proposed approach should be extended to allow the use of several alternative domain models instead of a single commonly agreed ontology.

6.2.6 Improved Service Description Derivation

To improve the OWL-S derivation approach, *results* indicating failures should also be considered. Currently, Service Monitor generates only process *results* representing successful service invocations. Nonetheless, the services may return special codes indicating a failure to fulfill the request, for example if the input parameters contain incorrect values. Currently, the failure cases can only be detected if none of the *results* of the executed OWL-S *process* corresponds to the perceived results of invoking a service.

6.2.7 Transparent Cloud Resource Reservation

A current limitation of the proposed approach is that cloud resources are somewhat inefficiently used. While it is possible to create a network of Cloud Gateway services residing on separate virtual machines, the machines must be leased in a static manner, before launching the corresponding Cloud Gateways. Cloud Gateway could be enhanced so that it dynamically created new virtual machines as the utilization of the existing ones reached a certain level.

Cloud Gateway measures the percentage of used memory and the system load average to avoid the overuse of computing resources. The method appears effective in preventing severe performance degradation when starting several applications. However, currently, Cloud Gateway is unable to estimate the amount of resources an application will consume once it has been started. Therefore, future research should target the implementation of a mechanism for evaluating the run-time resource consumption of the started applications.

In addition, this research work has experimented with web service Orchestration spanning separate computing clouds. The results suggest that using computing cloud resources causes no considerable performance drawbacks. However, deploying

several web services on separate virtual machines requires considerable effort. Future research should investigate the use of Cloud Gateway in automating this task.

6.3 Future Research Directions

A sophisticated AI planner appears a necessary prerequisite for efficient domain-independent service composition. While several such planners are already available, many of them are restricted by licenses. The development of new planners should involve both experimenting with the current planning algorithms and inventing completely new planning techniques.

REFERENCES

[1] ActiveBPEL™ Designer and Eclipse Web Tools Project. http://www.activebpel.org/samples/samples-3/eclipseWTP_and_BPEL/doc/index.html. Accessed: 2008-7-2.

[2] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M.-T. Schmidt, A. Sheth, and K. Verma. Web service semantics - WSDL-S. http://www.w3.org/Submission/WSDL-S/, 2005. Accessed: 2013-11-11.

[3] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C. K. Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web services business process execution language. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 2007. Accessed: 2013-11-11.

[4] BPEL Service Engine User's Guide. https://open-esb.dev.java.net/kb/preview3/ep-bpel-se.html. Accessed: 2008-4-25.

[5] D. Calvanese, G. D. Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Data complexity of query answering in description logics. *Artificial Intelligence*, 195(0):335–360, February 2013.

[6] D. Çelik and A. Elçi. A semantic search agent approach: Finding appropriate semantic web services based on user request term(s). In *ITI 3rd International Conference on Information and Communications Technology, ICICT 2005 - Enabling Technologies for the New Knowledge Society*, pages 675–688, 2005.

[7] D. Çelik and A. Elçi. Discovery and scoring of semantic web services based on client requirement(s) through a semantic search agent. In *Proceedings - International Computer Software and Applications Conference*, volume 2, pages 273–278, 2006.

[8] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web service definition language (WSDL). http://www.w3.org/TR/wsdl, 2001. Accessed: 2013-11-11.

[9] J. Clark. XSL Transformations (XSLT). http://www.w3.org/TR/xslt, 1999. Accessed: 2013-10-10.

[10] J. Clark and S. DeRose. XML Path Language (XPath). http://www.w3.org/TR/xpath/. Accessed: 2014-6-19.

[11] W. Dai and V. Vyatkin. Transformation from PLC to distributed control using ontology mapping. In *Industrial Informatics (INDIN), 2012 10th IEEE International Conference on*, pages 436–441, 2012.

[12] I. M. Delamer and J. L. Martinez Lastra. Ontology modeling of assembly processes and systems using semantic web services. In *Industrial Informatics, 2006 IEEE International Conference on*, pages 611–617, 16-18 Aug. 2006.

[13] I. M. Delamer and J. L. Martinez Lastra. Service-oriented architecture for distributed publish/subscribe middleware in electronics production. *Industrial Informatics, IEEE Transactions on*, 2(4):281–294, 2006.

[14] I. M. Delamer and J. L. Martinez Lastra. Loosely-coupled automation systems using device-level SOA. In *Industrial Informatics, 2007 5th IEEE International Conference on*, volume 2, pages 743–748, 2007.

[15] I. M. Delamer, J. L. Martinez Lastra, and O. Perez. An evolutionary algorithm for optimization of XML publish/subscribe middleware in electronics production. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 681–688, 2006.

[16] Description logics. `http://dl.kr.org/`, 05 2013. Accessed: 2013-5-27.

[17] OASIS Devices profile for web services (DPWS). `http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01`. Accessed: 2014-6-19.

[18] DPWS4J toolkit by Schneider Electric. `https://forge.soa4d.org/projects/dpws4j/`. Accessed: 2014-6-19.

[19] Y. Evchina, A. Dvoryanchikova, and J. L. Martinez Lastra. Ontological framework of context-aware and reasoning middleware for smart homes with health and social services. In *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*, pages 985–990, 2012.

[20] J. Farrell and H. Lausen. Semantic annotations for WSDL and XML Schema. `http://www.w3.org/TR/sawsdl/`, 2007. Accessed: 2014-6-19.

[21] D. Fensel and C. Bussler. The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2):113–137, Summer 2002.

[22] L. Ferrarini, C. Veber, A. Lüder, J. Peschke, A. Kalogeras, J. Gialelis, J. Rode, D. Wünsch, and V. Chapurlat. Control architecture for reconfigurable manufacturing systems: The PABADIS'PROMISE approach. In *IEEE Symposium on Emerging Technologies and Factory Automation, ETFA*, pages 545–552, 2006.

[23] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, 2008.

[24] B. C. Gazen and C. A. Knoblock. Combining the expressivity of UCPOP with the efficiency of Graphplan. In *Proceedings of the 4th European Conference on Planning: Recent Advances in AI Planning*, ECP '97, pages 221–233, London, UK, 1997. Springer-Verlag.

[25] P. Gearon, A. Passant, and A. Polleres. SPARQL 1.1 Update W3C Proposed Recommendation 08 November 2012. `http://www.w3.org/TR/sparql11-update/`. Accessed: 2013-11-11.

[26] R. Grønmo, M. C. Jaeger, and H. Hoff. *Transformations between UML and OWL-S*, volume 3748 of *Lecture Notes in Computer Science*. Springer, 2005.

[27] O. Hatzi, D. Vrakas, N. Bassiliades, D. Anagnostopoulos, and I. Vlahavas. Semantic awareness in automated web service composition through planning. In S. Konstantopoulos, S. Perantonis, V. Karkaletsis, C. Spyropoulos, and G. Vouros, editors, *Artificial Intelligence: Theories, Models and Applications*, volume 6040, pages 123–132. Springer Berlin / Heidelberg, 2010.

[28] G. C. Hobold and F. Siqueira. Discovery of semantic web services compositions based on SAWSDL annotations. In *Web Services (ICWS), 2012 IEEE 19th International Conference on*, pages 280–287, June 2012.

[29] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:263–312, 2001.

[30] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A semantic web rule language combining OWL and RuleML. `http://www.w3.org/Submission/SWRL/`.

[31] A. Hristoskova, D. Moeyersoon, S. V. Hoecke, S. Verstichel, J. Decruyenaere, and F. D. Turck. Dynamic composition of medical support services in the ICU: Platform and algorithm design details. *Computer methods and programs in biomedicine*, 100(3):248–264, December 2010.

[32] IEC 61131-3 programmable controllers - part 3: Programming languages, 1993.

[33] IEC 61499-1 Function Blocks - Architecture, 2005.

[34] K. Iqbal, M. L. Sbodio, V. Peristeras, and G. Giuliani. Semantic service discovery using SAWSDL and SPARQL. In *International Conference on Semantics, Knowledge and Grid*, pages 205–212, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[35] F. Jammes and H. Smit. Service-oriented paradigms in industrial automation. *Industrial Informatics, IEEE Transactions on*, 1(1):62–70, 2005.

[36] F. Jammes, H. Smit, J. L. Martinez Lastra, and I. M. Delamer. Orchestration of service-oriented manufacturing processes. In *Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on*, volume 1, pages 8 pp.–624, 2005.

[37] Reasoners and rule engines: Jena inference support. `http://jena.apache.org/documentation/inference/index.html`. Accessed: 2013-2-1.

[38] P. Jiang, X. Shao, L. Gao, H. Qiu, and P. Li. A process-view approach for cross-organizational workflows management. *Advanced Engineering Informatics*, 24(2):229–240, 2010.

[39] A. P. Kalogeras, J. V. Gialelis, C. E. Alexakos, M. J. Georgoudakis, and S. A. Koubias. Vertical integration of enterprise industrial systems utilizing web services. *Industrial Informatics, IEEE Transactions on*, 2(2):120–128, 2006.

[40] K. Keahey, M. Tsugawa, A. Matsunaga, and J. A. B. Fortes. Sky computing. *Internet Computing, IEEE*, 13(5):43–51, 2009.

[41] I.-W. Kim and K.-H. Lee. Describing semantic web services: From UML to OWL-S. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 529–536, 2007.

[42] I.-W. Kim and K.-H. Lee. A model-driven approach for describing semantic web services: From UML to OWL-S. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 39(6):637–646, 2009.

[43] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C Recommendation 10 February 2004. `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`. Accessed: 2014-6-19.

[44] T. A. Kvaløy, E. Rongen, A. Tirado-Ramos, and P. Sloot. Automatic composition and selection of semantic web services. *Lecture Notes in Computer Science*, 3470:266–271, 2005.

[45] G. Lawton. Developing software online with platform-as-a-service technology. *Computer*, 41(6):13–15, 2008.

[46] C. Legat, D. Schütz, and B. Vogel-Heuser. Automatic generation of field control strategies for supporting (re-)engineering of manufacturing systems. *Journal of Intelligent Manufacturing*, pages 1–11, 03/03 2013.

[47] P. Leitão and N. Rodrigues. Modelling and validating the multi-agent system behaviour for a washing machine production line. In *Industrial Electronics (ISIE), 2012 IEEE International Symposium on*, pages 1203–1208, 2012.

[48] A. Lobov, J. Puttonen, V. Villasenor Herrera, R. Andiappan, and J. L. Martinez Lastra. Service oriented architecture in developing of loosely-coupled manufacturing systems. In *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*, pages 791–796, July 2008.

[49] A. Lobov, F. Ubis Lopez, V. Villasenor Herrera, J. Puttonen, and J. L. Martinez Lastra. Semantic web services framework for manufacturing industries. In *Robotics and Biomimetics, 2008. ROBIO 2008. IEEE International Conference on*, pages 2104–2108, February 2009.

[50] O. J. López Orozco and J. L. Martinez Lastra. A real-time interface for agent-based control. In *Industrial Embedded Systems, 2007. SIES '07. International Symposium on*, pages 49–54, 2007.

[51] M. Loskyll, J. Schlick, S. Hodek, L. Ollinger, T. Gerber, and B. Pirvu. Semantic service discovery and orchestration for manufacturing processes. In *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–8, 2011.

[52] P. Louridas. Orchestrating web services with BPEL. *Software, IEEE*, 25(2):85–87, 2008.

[53] D. Martin, M. Burstein, D. McDermott, S. McIlraith, M. Paolucci, K. Sycara, D. L. McGuinness, E. Sirin, and N. Srinivasan. Bringing semantics to web services with OWL-S. *World Wide Web*, 10(3):243–277, 07/ 2007.

[54] D. Martin, M. Paolucci, and M. Wagner. *Bringing Semantic Annotations to Web Services: OWL-S from the SAWSDL Perspective*, volume 4825 of *Lecture Notes in Computer Science*, pages 340–352. Springer Berlin / Heidelberg, 2007.

[55] P. Martinek, B. Tothfalussy, and B. Szikora. Implementation of semantic services in enterprise application integration. *WSEAS Transactions on Computers*, 7(10):1658–1668, 2008.

[56] J. L. Martinez Lastra and I. Delamer. Ontologies for production automation. In T. Dillon, E. Chang, R. Meersman, and K. Sycara, editors, *Advances in Web Semantics I*, volume 4891 of *Lecture Notes in Computer Science*, pages 276–289. Springer, 2008.

[57] J. L. Martinez Lastra and I. M. Delamer. Semantic web services in factory automation: fundamental insights and research roadmap. *IEEE Transactions on Industrial Informatics*, 2(1):1–11, 2006.

[58] J. L. Martinez Lastra and I. M. Delamer. *Ontologies for production automation*, volume 4891 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer, 2008.

[59] J. L. Martinez Lastra, L. Godinho, A. Lobov, and R. Tuokko. An IEC 61499 application generator for scan-based industrial controllers. In *Industrial Informatics, 2005. INDIN '05. 2005 3rd IEEE International Conference on*, pages 80–85, 2005.

[60] D. McDermott. PDDL - the planning domain definition language version 1.2. Technical report, Yale Center for Computational Vision and Control, 1998.

[61] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. `http://www.w3.org/TR/owl-features/`, 2004. Accessed: 2013-11-11.

[62] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic web services. *Intelligent Systems, IEEE*, 16(2):46–53, 2001.

[63] J. M. Mendes, A. Bepperling, J. Pinto, P. Leitao, F. Restivo, and A. W. Colombo. Software methodologies for the engineering of service-oriented industrial automation: The continuum project. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 1, pages 452–459, 2009.

[64] T. Moser and S. Biffl. Semantic integration of software and systems engineering environments. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 42(1):38–50, 2012.

[65] J. Nitzsche, T. V. Lessen, D. Karastoyanova, and F. Leymann. Bpel for semantic web services (bpel4sws). In *Proceedings of the 2007 OTM confederated international conference on On the move to meaningful internet systems - Volume Part I*, OTM'07, pages 179–188, Berlin, Heidelberg, 2007. Springer-Verlag.

[66] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*, pages 124–131, 2009.

[67] OASIS Consortium. `https://www.oasis-open.org/`. Accessed: 2013-5-15.

[68] OASIS Web Services Dynamic Discovery (WS-Discovery). `http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01`. Accessed: 2013-1-22.

[69] M. J. O'Connor and A. K. Das. SQWRL: A Query Language for OWL. In *OWLED*, 2009.

[70] A. L. Opdahl. Semantic annotations for modelling language interoperability. In *ACM International Conference Proceeding Series*, 2011.

[71] OWL-S. OWL for Services (OWL-S) Home Page. `http://www.ai.sri.com/daml/services/owl-s/`. Accessed: 2013-11-11.

[72] OWL-S API. `http://www.mindswap.org/2004/owl-s/api/index.shtml`. Accessed: 2013-5-15.

[73] M. Paolucci, A. Ankolekar, N. Srinivasan, and K. Sycara. The DAML-S virtual machine. *Lecture Notes in Computer Science*, 2870:290–305, 2003.

[74] M. Paolucci, M. Wagner, and D. Martin. Grounding owl-s in sawsdl. In *Proceedings of the 5th international conference on Service-Oriented Computing*, pages 416–421, 2007.

[75] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[76] B. Pi, G. Zou, C. Zhong, J. Zhang, H. Yu, and A. Matsuo. Flow editor: Semantic web service composition tool. In *Proceedings - 2012 IEEE 9th International Conference on Services Computing, SCC 2012*, pages 666–667, 2012.

[77] C. Popescu, A. Lobov, J. L. Martinez Lastra, and M. A. C. Soto. A modelling approach to formally represent service orchestration. *International Journal of Computer Aided Engineering and Technology*, 1(1):1–30, 01/01 2008.

[78] The Protégé Ontology Editor and Knowledge Acquisition System. `http://protege.stanford.edu/`. Accessed: 2013-5-15.

[79] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`, 2008. Accessed: 2013-11-11.

[80] J. Puttonen, A. Lobov, M. A. Cavia Soto, and J. L. Martinez Lastra. A semantic web services-based approach for production systems control. *Advanced Engineering Informatics*, 24(3):285–299, August 2010.

[81] J. Puttonen, A. Lobov, and J. L. Martinez Lastra. An application of BPEL for service orchestration in an industrial environment. *2008 IEEE International Conference on Emerging Technologies and Factory Automation, Proceedings*, pages 530–537, 2008.

[82] J. Puttonen, A. Lobov, and J. L. Martinez Lastra. An approach to service deployment to the service cloud. In *ICONS 2011 : The Sixth International Conference on Systems*, pages 122–127, from January 23 to January 28 2011.

[83] J. Puttonen, A. Lobov, and J. L. Martinez Lastra. Maintaining a dynamic view of semantic web services representing factory automation systems. In *2013 IEEE 20th International Conference on Web Services*, pages 419–426, 27 June-2 July 2013.

[84] J. Puttonen, A. Lobov, and J. L. Martinez Lastra. Semantics-based composition of factory automation processes encapsulated by web services. *Industrial Informatics, IEEE Transactions on*, 9(4):2349–2359, 2013.

[85] J. Rao and X. Su. A survey of automated web service composition methods. In *In Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition, SWSWPC 2004*, pages 43–54, 2004.

[86] RDF/XML Syntax Specification (Revised), W3C Recommendation 10 February 2004. `http://www.w3.org/TR/REC-rdf-syntax/`. Accessed: 2014-6-19.

[87] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web service modeling ontology. *Applied Ontology*, 1(1):77–106, 2005.

[88] A. Sasa, M. B. Juric, and M. Krisper. Service-oriented framework for human task support and automation. *Industrial Informatics, IEEE Transactions on*, 4(4):292–302, 2008.

[89] M. L. Sbodio, D. Martin, and C. Moulin. Discovering semantic web services using SPARQL and intelligent agents. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(4):310–328, November 2010.

[90] A. Segev and Q. Sheng. Bootstrapping ontologies for web services. *Services Computing, IEEE Transactions on*, 5(1):33–44, Jan.-March 2012.

[91] S. Sellami and O. Boucelma. Web services discovery and composition: A schema matching approach. In *Web Services (ICWS), 2011 IEEE International Conference on*, pages 706–707, 2011.

[92] N. Shah, K. M. Chao, A. N. Godwin, and A. James. An abstract knowledge based approach to diagnosis and recovery of plan failure in multi-agent systems. *Advanced Engineering Informatics*, 21(2):183–190, 2007.

[93] A. Sheth and A. Ranabahu. Semantic modeling for cloud computing, part 1. *Internet Computing, IEEE*, 14(3):81–83, 2010.

[94] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN planning for web service composition using SHOP2. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):377–396, October 2004.

[95] Socrades project. `http://wwww.socrades.eu`. Accessed: 2012-2-21.

[96] T. X. Song, P. J. Tian, Y. H. Liu, and B. Q. Huang. Web services' semantic annotation and auto-matching based on sawsdl. In *2008 International Symposium on Information Science and Engineering, ISISE 2008*, volume 2, pages 577–580, 2008.

[97] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. *Internet Computing, IEEE*, 13(5):14–22, 2009.

[98] M. Spranger, C. Thiele, and M. Hild. Integrating high-level cognitive systems with sensorimotor control. *Advanced Engineering Informatics*, 24(1):76–83, 2010.

[99] K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan. Automated discovery, interaction and composition of semantic web services. *Web Semantics*, 1(1):27–46, 2003.

[100] P. S. Tan, A. E. S. Goh, and S. S. G. Lee. An ontology to support context-aware b2b services. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 586–593, 2010.

[101] S. tao Sun, D. sheng Liu, G.-Q. Li, W. yang Yu, and L. Pang. The research on hierarchical construction method of domain ontology. In *Semantics Knowledge and Grid (SKG), 2010 Sixth International Conference on*, pages 203–210, 2010.

[102] J. T. E. Timm and G. C. Gannod. A model-driven approach for specifying semantic web services. In *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*, pages 313–320 vol.1, 2005.

[103] J. T. E. Timm and G. C. Gannod. Grounding and execution of OWL-S based semantic web services. In *Services Computing, 2008. SCC '08. IEEE International Conference on*, volume 2, pages 588–592, 2008.

[104] H. Tong, J. Cao, S. Zhang, and M. Li. A distributed algorithm for web service composition based on service agent model. *IEEE Transactions on Parallel and Distributed Systems*, 22(12):2008–2021, 2011.

[105] UDDI. `http://uddi.xml.org/`. Accessed: 2013-5-15.

[106] M. K. Uddin, A. Dvoryanchikova, A. Lobov, and J. L. Martinez Lastra. An ontology-based semantic foundation for flexible manufacturing systems. In *IECON 2011 - 37th Annual Conference on IEEE Industrial Electronics Society*, pages 340–345, 2011.

[107] M. K. Uddin, J. Puttonen, S. Scholze, A. Dvoryanchikova, and J. L. Martinez Lastra. Ontology-based context-sensitive computing for FMS optimization. *Assembly Automation*, 32(2):163–174, 2012.

[108] University of Basel. ON :: OWL-S API - Introduction. `http://on.cs.unibas.ch/owls-api/index.html`. Accessed: 2013-11-11.

[109] R. Vaculín and K. Sycara. Semantic web services monitoring: An owl-s based approach. In *Proceedings of the Annual Hawaii International Conference on System Sciences*, 2008.

[110] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput.Commun.Rev.*, 39(1):50–55, dec 2008.

[111] K. Verma. Configuration and adaptation of semantic web processes. `http://lsdis.cs.uga.edu/library/download/thesis/kunal/verma_kunal_200608_phd.pdf`, 2006. Accessed: 2013-11-11.

[112] V. Villaseñor Herrera, A. Vidales Ramos, and J. L. Martinez Lastra. An agent-based system for orchestration support of web service-enabled devices in discrete manufacturing systems. *Journal of Intelligent Manufacturing*, 23(6):2681–2702, Mon, 16 May 2012.

[113] R. D. Virgilio. Meta-modeling of semantic web services. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 162–169, 2010.

[114] V. V. Vyatkin, J. H. Christensen, and J. L. Martinez Lastra. OOONEIDA: An open, object-oriented knowledge economy for intelligent industrial automation. *IEEE Transactions on Industrial Informatics*, 1(1):4–16, 2005.

[115] R. Walker. Examining load average. `http://www.linuxjournal.com/article/9001`. Accessed: 2013-5-14.

[116] H. H. Wang, N. Gibbins, T. R. Payne, and D. Redavid. A formal model of the semantic web service ontology (wsmo). *Information Systems*, 37(1):33–60, March 2012.

[117] G. Wasson and M. Humphrey. Exploiting WSRF and WSRF.NET for remote job execution in grid environments. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 12a–12a, 2005.

[118] Web services choreography description language version 1.0. `http://www.w3.org/TR/ws-cdl-10/`. Accessed: 2013-5-16.

[119] Web Services Eventing (WS-Eventing). `http://www.w3.org/Submission/WS-Eventing/`.

[120] D16 The WSML Specification. `http://www.wsmo.org/TR/d16/`, 2008. Accessed: 2013-11-11.

[121] OASIS Web Services Resource Framework (WSRF) TC. `http://www.oasis-open.org/committees/wsrf/`. Accessed: 2013-5-15.

[122] XMI. `http://www.omg.org/spec/XMI/`. Accessed: 2013-5-15.

[123] B. Yang and Z. Qin. Composing semantic web services with PDDL. *Information Technology Journal*, 9(1):48–54, 2010.

[124] J.-H. Yang and I.-J. Chung. A method for automatic generation of owl-s service ontology. *International Journal of Information Processing System*, 2(2):114–114–123, 2006.

[125] S. H. Yeganeh, J. Habibi, H. Rostami, and H. Abolhassani. Semantic web service composition testbed. *Computers & Electrical Engineering*, 36(5):805–817, 9 2010.

[126] D. Yulin and Z. Chunjiao. Design and research of embedded PLC development system. In *Computer Research and Development (ICCRD), 2011 3rd International Conference on*, volume 3, pages 226–228, 2011.

[127] Q. Zhu. Topologies of agents interactions in knowledge intensive multi-agent systems for networked information services. *Advanced Engineering Informatics*, 20(1):31–45, 2006.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland

ISBN 978-952-15-3315-0
ISSN 1459-2045