*i*SEMSERV:
A FRAMEWORK FOR ENGINEERING INTELLIGENT SEMANTIC SERVICES

Jabu Saul Mtsweni

Submitted in accordance with the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in the subject

COMPUTER SCIENCE

at the

UNIVERSITY OF SOUTH AFRICA

Supervisor: **PROF. E. BIERMANN**
Co-Supervisor: **PROF. L. PRETORIUS**

January 2013

DECLARATION

Student number: **46159525**

I, **Jabu Saul Mtsweni** declare that the thesis titled **iSEMSERV: A FRAMEWORK FOR ENGINEERING INTELLIGENT SEMANTIC SERVICES** is my own work and all the sources that have been used and quoted are indicated and acknowledged by means of complete references.

Signature : [illegible]

Date : **21 January 2013**

DEDICATION

This study is dedicated to my late uncle

Thomas Peter Masango,

for his courage and support throughout my tertiary education.

ACKNOWLEDGEMENTS

My deepest and humble gratitude goes to my Father in Heaven, without Whom it would not have been possible to achieve this milestone.

I further wish to express my deepest love and appreciation to my family, particularly my wife (Martha), and our two children (Ziyanda and Anelisa). Their support and sacrifices are always appreciated.

To my supervisors, Prof. Elmarie Biermann and Prof. Laurette Pretorius, I am grateful for your patience, expert guidance, and above all, the freedom you allowed me to shape this thesis according to my capabilities.

I am greatly thankful to SAP Research, CSIR Meraka, University of South Africa, National Research Foundation (NRF), and SANPAD (RCI Programme) for all the support (financial and otherwise). I would also like to sincerely thank all the professionals, pastors, friends, relatives, colleagues, and students who contributed and supported me in different ways during the course of this study; if I could list all the names, it would amount to another thesis.

Thank you all, and God bless!

ABSTRACT

The need for modern enterprises and Web users to simply and rapidly develop and deliver platform-independent services to be accessed over the Web by the global community is growing. This is self-evident, when one considers the omnipresence of electronic services (e-services) on the Web.

Accordingly, the Service-Oriented Architecture (SOA) is commonly considered as one of the de facto standards for the provisioning of heterogeneous business functionalities on the Web. As the basis for SOA, Web Services (WS) are commonly preferred, particularly because of their ability to facilitate the integration of heterogeneous systems. However, WS only focus on syntactic descriptions when describing the functional and behavioural aspects of services. This makes it a challenge for services to be automatically discovered, selected, composed, invoked, and executed – without any human intervention. Consequently, Semantic Web Services (SWS) are emerging to deal with such a challenge.

SWS represent the convergence of Semantic Web (SW) and WS concepts, in order to enable Web services that can be automatically processed and understood by machines operating with limited or no user intervention. At present, research efforts within the SWS domain are mainly concentrated on semantic services automation aspects, such as discovery, matching, selection, composition, invocation, and execution. Moreover, extensive research has been conducted on the conceptual models and formal languages used in constructing semantic services.

However, in terms of the engineering of semantic services, a number of challenges are still prevalent, as demonstrated by the lack of development and use of semantic services in real-world settings. The lack of development and use could be attributed to a number of challenges, such as complex semantic services enabling technologies, leading to a steep learning curve for service developers; lack of unified service platforms for guiding and supporting simple and rapid engineering of semantic services, and the limited integration of semantic technologies with mature service-oriented technologies.

In addition, a combination of isolated software tools is normally used to engineer semantic services. This could, however, lead to undesirable consequences, such as prolonged service development times, high service development costs, lack of services re-use, and the lack of semantics interoperability, reliability, and re-usability. Furthermore, available software platforms do not support the creation of semantic services that are intelligent beyond the application of semantic descriptions, as envisaged for the next generation of services, where the connection of knowledge is of core importance.

In addressing some of the challenges highlighted, this research study adopted a qualitative research approach with the main focus on conceptual modelling. The main contribution of this study is thus a framework called *i*SemServ to simplify and accelerate the process of engineering intelligent semantic services. The framework has been modelled and developed, based on the principles of simplicity, rapidity, and intelligence. The key contributions of the proposed framework are: (1) An end-to-end and unified approach of engineering intelligent semantic services, thereby enabling service engineers to use one platform to realize all the modules comprising such services; (2) proposal of a model-driven approach that enables the average and expert service engineers to focus on developing intelligent semantic services in a structured, extensible, and platform-independent manner. Thereby increasing developers' productivity and minimizing development and maintenance costs; (3) complexity hiding through the exploitation of template and rule-based automatic code generators, supporting different service architectural styles and semantic models; and (4) intelligence wrapping of services at message and knowledge levels, for the purposes of automatically processing semantic service requests, responses and reasoning over domain ontologies and semantic descriptions by keeping user intervention at a minimum.

The framework was designed by following a model-driven approach and implemented using the Eclipse platform. It was evaluated using practical use case scenarios, comparative analysis, and performance and scalability experiments. In conclusion, the *i*SemServ framework is considered appropriate for dealing with the complexities and restrictions involved in engineering intelligent semantic services, especially because the amount of time required to generate intelligent semantic services using the proposed framework is smaller compared with the time that the service engineer would need to manually generate all the different artefacts comprising an intelligent semantic service.

Keywords: Intelligent semantic services, Web services, Ontologies, Intelligent agents, Service engineering, Model-driven techniques, *i*SemServ framework.

TABLE OF CONTENTS

I. LIST OF FIGURES

II. LIST OF TABLES

III. LIST OF LISTINGS

IV. LIST OF ABBREVIATIONS

ACL	Agent Communication Language
AI	Artificial Intelligence
API	Application Programming Interface
BPD	Business Process Diagram
BPMN	Business Process Modelling Notation
CDE	Code-driven Engineering
CMU	Carnegie Mellon University
DAML	DARPA Agent Mark-up Language
DAML-S	DARPA Agent Mark-up Language for Services
DIP	Data, Information, Processes, Integration with SWS
DL	Description Logics
EMF	Eclipse Modelling Framework
EU	European Union
FIPA	Foundation for Intelligent Physical Agents
FOL	First Order Logic
FP7	Framework Programme Seven
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IIOP	Internet Inter-ORB Protocol
ILEV	Intelligence Logic Editor and Validator
IRS-III	Internet Reasoning Service
ISEMSERV	Intelligent Semantic Service (framework)
ISS	Intelligent Semantic Service (service)
JADE	Java Agent Development Environment
JAX-RS	Java API for RESTful Web Services
JAX-WS	Java API for XML Web Services
JEE	Java Enterprise Edition
JESS	Java Expert System Shell
LP	Logic Programming
MAS	Multi-Agent System
MDA	Model-driven Architecture
MDE	Model-driven Engineering
MOF	Meta-Object Facility
MTL	Model-to-Text Language
OCML	Operational Conceptual Modelling Language
OMG	Object Management Group
OSS	Open Source Software
OWL	Web Ontology Language
OWL-S	Web Ontology Language for Services
PEOU	Perceived Ease of Use
POJO	Plain Old Java Object

PIM	Platform Independent Model
PML	Proof Mark-up Language
PSM	Platform Specific Model
PU	Perceived Usefulness
QoS	Quality of Service
RAD	Rapid Application Development
RDF	Resource description Framework
RDF-S	Resource Description Framework Schemas
REST	Representational State Transfer
RIF	Rules Interchange Format
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RULEML	Rule Mark-up Language
SASS	Service Architectural Style Selector
SAWSDL	Semantic Annotations for WSDL and XML Schema
SDK	Software Development Kit
SDLC	Software Development Lifecycle
SE	Software Engineering
SEALS	Semantic Evaluation at Large Scale
SEE	Semantic Execution Environment
SEMMAS	Semantic Web Services and Multi-Agent System
SMTP	Simple Mail Transfer Protocol
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SOSE	Service-Oriented Software Engineering
SPA	Service Provider Agent
SPARQL	Simple Protocol & RDF Query Language
SQL	Structured Query Language
SRA	Service Request Agent
SVEV	Semantics Visualizer/Editor/Validator
SW	Semantic Web
SWF	Semantic Web Fred
SWS	Semantic Web Services
SWSA	Semantic Web Services Architecture
SWSL	Semantic Web Services Language
TAM	Technology Adoption Model
UDDI	Universal Description Discovery and Integration
UI	User Interfaces
UML	Unified Modelling Language
UML-S	Unified Modelling Language for Services
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WADL	Web Application Description Language
WE	Web Engineering

WEBML	Web Modelling Language
WS	Web Services
WSDL	Web Service Description Language
WSDL-S	Web Service Description Language for Semantics
WSML	Web Service Modelling Language
WSMO	Web Service Modelling Ontology
WSMX	Web Service Execution Environment
WWW	World Wide Web
XML	Extensible Mark-up Language
XSD	XML Schema Definition

CHAPTER 1: Proposal

This chapter introduces the challenges addressed in this thesis. In attempting to address the identified problems, the discussions delve into the main research question and subsidiary questions, the research objectives, and the research methodologies applied to propose the appropriate solution that could address the challenges identified. In addition, the evaluation techniques for the proposed solution are described. Lastly, the primary and secondary contributions emanating from this study are enumerated.

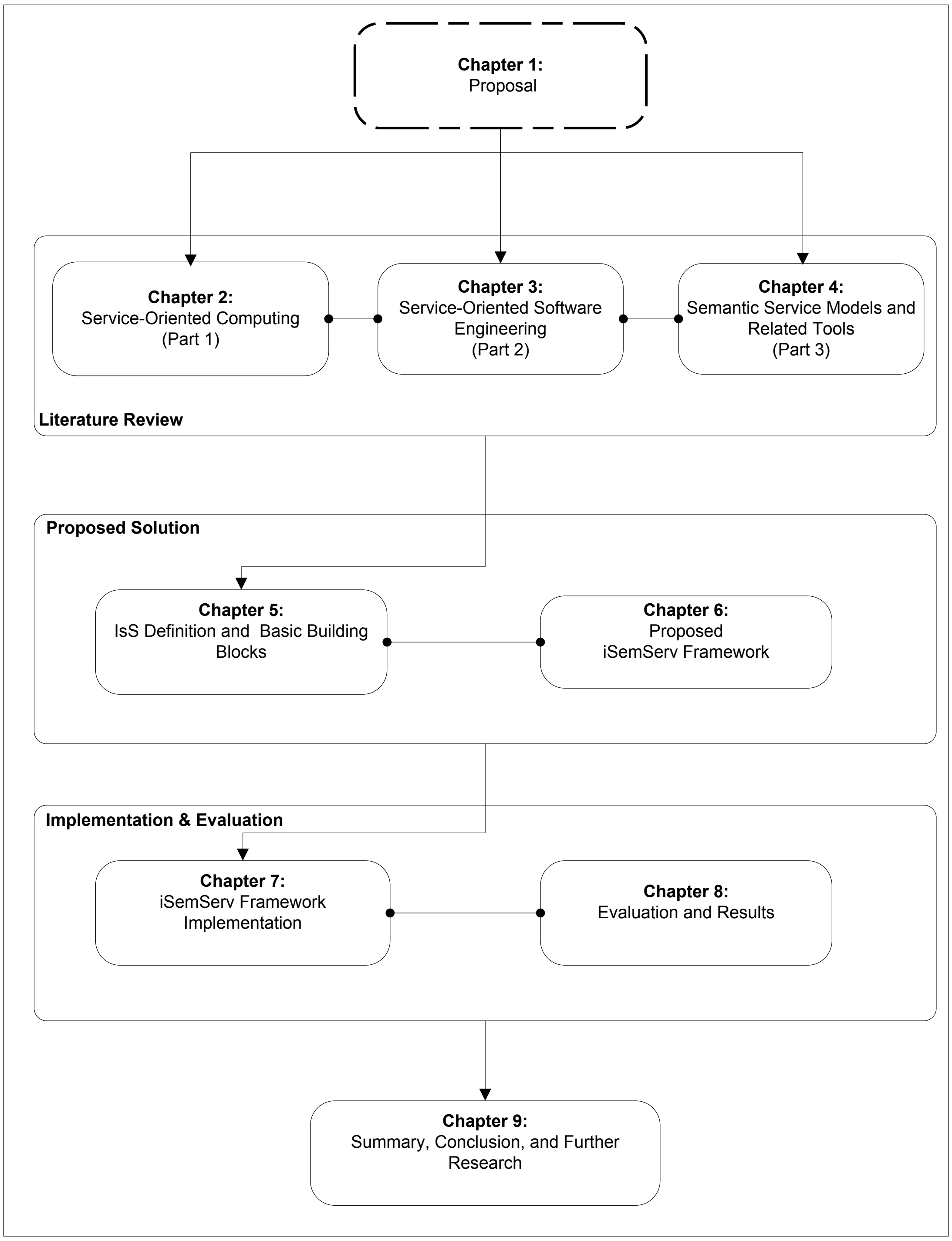

Chapter 1:
Proposal
Chapter 2:
Service-Oriented Computing
(Part 1)
Chapter 3:
Service-Oriented Software Engineering
(Part 2)
Chapter 4:
Semantic Service Models and Related Tools
(Part 3)
Literature Review
Proposed Solution
Chapter 5:
IsS Definition and Basic Building Blocks
Chapter 6:
Proposed iSemServ Framework
Implementation & Evaluation
Chapter 7:
iSemServ Framework Implementation
Chapter 8:
Evaluation and Results
Chapter 9:
Summary, Conclusion, and Further Research

Figure 1.1: Overall Thesis Structure

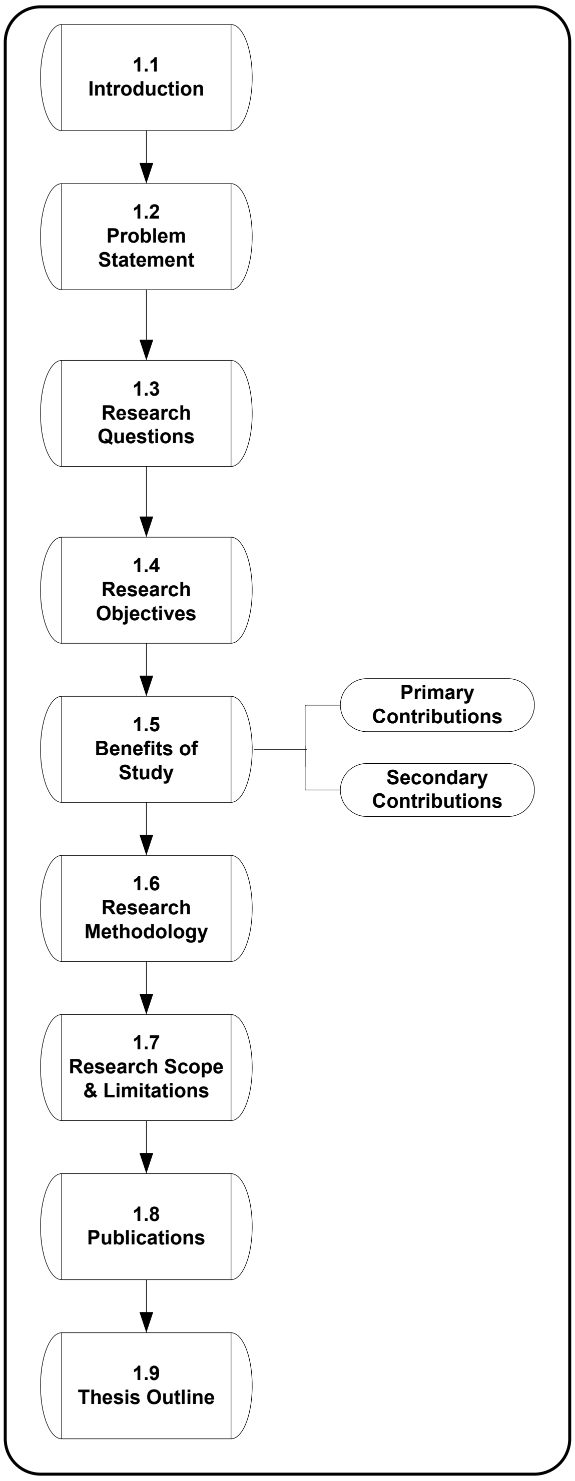


Figure 1.2: Chapter 1 Layout

1.1. INTRODUCTION

The Web has evolved into a universal virtual environment, where distributed applications and business services are published and consumed (Sheng *et al.*, 2010). In addition, the interoperability of Web services (WS) with legacy systems has revolutionized the exposition and consumption of business processes on the Web. This is demonstrated by the widespread adoption of the Service-Oriented Architecture (SOA) by prominent global enterprises (Hassanzadeh, Namdarian & Elahi, 2011; Hemayati *et al.*, 2010; Stein., 2008).

The terms *business service, e-service, and Web service* are generally interpreted differently in varying contexts. In this study, we define the term *business service* as any useful business functionality that is provided or requested via any appropriate means to capture value for both the consumer and the provider (Baida, Gordijn & Omelayenko, 2004), such as, for example, opening a bank account.

Accordingly, an *e-service* is defined as the provision of a business service via any type of electronic network (Sun & Lau, 2007:365), such as the Internet. For instance, buying books via the Internet can be regarded as a form of e-service. In this study, we refer to e-services and Web services, as related but different concepts.

According to Booth *et al*. (2004), *Web services* are software components that provide basic standards to enable interoperability between different software applications, running on different platforms. They are software components available in a distributed environment; they perform business-specific functions and facilitate the integration of disparate software systems. Alonso *et al.*(2004:124) clearly define a Web service as a: *"software application identified by a URI*[1]*, whose interface and bindings are capable of being defined, described, advertised, discovered, and invoked".*

In the context of this study, a Web service is a component that comprises specific business functionality that could be accessed over the Web; and an e-service is a collection of network-resident services (Cardoso, Voigt & Winkler, 2008).

[1] URI stands for Uniform Resource Identifier

From the academic research perspective, it is apparent that WS have transformed the World Wide Web (WWW) from being a source of raw data and information to a platform of distributed services (Filho & Ferreira, 2009; García-Sanchez, 2007). However, since the main goal of WS is to facilitate worldwide accessibility of business services (Shen *et al.*, 2005) on the Web, issues of automatic deployment, discovery, invocation, and the composition of services are not addressed within the WS paradigm (Bensaber & Malki, 2008; Shen *et al.*, 2005). The main reason for this is that WS lack semantic descriptions that could facilitate services automation (Cabral *et al.*, 2004). Semantic descriptions are formal and rich annotations that unequivocally describe the non-functional, functional, and behavioural aspects of services (Stollberg, Hepp & Fensel, 2010).

The evolution of *Semantic Web Services* (SWS), sometimes generally referred to as *semantic services,* is bringing forth services that could be interpreted and processed by both humans and machines(Lu, Zhang & Ruan, 2007) – subsequently, enabling services to be integrated and utilized with little or no human intervention (Corcho *et al.*, 2007). *Semantic services* are commonly defined as extensions to the capabilities of WS by leveraging new and existing Web services with semantic descriptions – to facilitate and support automatic discovery, invocation, composition, and the execution of services (Bensaber & Malki, 2008; Janev & Vranes, 2010).

SWS are mainly evolving from the Semantic Web (SW) domain. SW is an extension of the current Web, where semantic annotations are incorporated into the current Web to facilitate machine-to-human communication and machine-to-machine communication (Berners-Lee, Hendler & Lassila, 2001; Rebstock, 2009:145). The benefits of SWS are well documented in the literature and some of the common benefits include, but are not limited to: (1) Improved representation, sharing, searching, reasoning, and the re-use of data and services; (2) anywhere and anytime dynamic connection of business partners; (3) the automation of various tasks on the Web, such as service discovery; and (4) on-the-fly interoperation of heterogeneous software systems (Bachlechner, 2008; Janev & Vranes, 2010; Joo, 2011; McIlraith, Son & Zeng, 2001; Mtsweni, Biermann & Pretorius, 2010).

Research efforts within the SWS domain have mainly focused on specific automation aspects for services, such as discovery, selection, composition, invocation, and execution (Gomez-Perez, Gonzalez-Cabero & Lama, 2004; Kanellopoulos & Kotsiantis, 2006; Toch, Reinhartz-Berger & Dori, 2011; Yeganeh *et al.*, 2010). In addition, extensive research has been conducted on concepts, such as the emerging Web Services Modelling Ontology (WSMO) (Lara *et al.*, 2005),as well as formal languages for constructing semantic services and applications (Dimitrov *et al.*, 2007).

However, standard tools and integrated platforms that aim to simplify the engineering of semantic services are still lacking. This could probably be attributed to the fact that the SWS domain is still in its infancy and most research is concentrating on aspects that demonstrate the automation aspects of semantic services.

In the following section, we shall succinctly discuss the research problem and the motivations behind the proposed study.

1.2. PROBLEM STATEMENT

According to Agre *et al.* (2007) and Bensaber and Malki (2008), SWS are seldom adopted and utilized, despite their attractive promises. The lack of adoption and usage is attributed to a number of challenges (Filho & Ferreira, 2009; Janev & Vranes, 2010; Siorpaes & Simperl, 2010). For instance, developing semantic services is resource intensive, tedious, and complex, especially without the support of unified and effective development tools (Filho & Ferreira, 2009; Janev & Vranes, 2010; Kerrigan *et al.*, 2007).

Nevertheless, the concept of technology adoption is well researched. Davis, Bagozzi, and Warshaw (1989:982) proposed a Technology Adoption Model (TAM) that could be used to understand what influences end-users to adopt and use a particular technology. The TAM model suggests that perceived usefulness (PU) and perceived ease-of-use (PEOU) are the important factors that determine one's behavioural intention to adopt and use a particular technology (Davis, Bagozzi & Warshaw, 1989:985; Wahid, 2007:2). This implies that if a technology is useful and is easier to use, the chances of its adoption and usage by end-users are high.

Usefulness in this context is defined as the “prospective user’s subjective probability that using a specific application system will increase job performance”; and *ease of use* is defined as the “degree to which the end-user expects to use the system with minimal effort” (Davis, Bagozzi & Warshaw, 1989:985). TAM also suggests that external factors, such as system design characteristics, system development processes, and system implementation processes may affect system adoption and usage.

Some of the other reasons for the infrequent adoption and usage of semantic services is the lack of unified platforms that are meant to simplify and accelerate the process of engineering semantic services (Bachlechner, 2008; Bensaber & Malki, 2008; Dimitrov *et al.*, 2007; Elenius *et al.*, 2005; Filho & Ferreira, 2009; Siorpaes & Simperl, 2010). Moreover, without the supporting tools, methods, and platforms, developers are hindered by the extra costs of manually adding semantic descriptions to new and existing services (Brambilla *et al.*, 2006; Filho & Ferreira, 2009; Janev & Vranes, 2010).

Currently, a combination of isolated software tools could be used to engineer semantic services. This could, however, lead to undesirable consequences, such as long service development times, high service development costs (Kerrigan *et al.*, 2007), the lack of semantic services re-use (Agarwal *et al.*, 2005; Filho & Ferreira, 2009), and the lack of reliability and re-usability of semantic descriptions (Siorpaes & Simperl, 2010). Furthermore, existing tools and platforms do not support the engineering of semantic services that are intelligent beyond the use of ontologies (Mtsweni, Biermann & Pretorius, 2010). As a result, there is a lack of semantic services that could be automatically processed and understood by machines with minimal user intervention.

Nonetheless, service developers cannot be expected to deliver error-free semantic services without relying on reliable and simple design and development methods and tools (Papazoglou & van den Heuvel, 2006). Furthermore, useful and easy-to-use tools, methods, and unified platforms are essential for a wider adoption (Dimitrov *et al.*, 2007) of emerging technologies, such as semantic services.

Consequently, the main aim of this study is to investigate and propose a service creation framework to simplify and accelerate the process of engineering intelligent semantic services. This includes implementing a proof-of-concept semantic service engineering platform and integrating emerging semantic technologies with matured and expansive Web service technologies. This also includes devising strategies for wrapping semantic services with intelligence, in order to realize service automation on the Web.

1.3. RESEARCH QUESTIONS

The overall thesis is based on the following main research question and supporting questions:

Main research question: *How could a unified service creation framework simplify and accelerate the process of engineering intelligent semantic services (IsS[2])?*

The following is a list of the supporting questions that are addressed in this study to exhaustively and satisfactorily answer the main research question stated above.

SQ[3]1: What are the fundamental building blocks and characteristics that constitute an *IsS*? *The notion of IsS is emerging, and with this particular supporting question, we attempt to understand and provide clarity as to what an IsS is, how an IsS is distinct from WS and SWS, and what components (building blocks) make up an IsS.*

SQ2: How could intelligent semantic services be developed from the identified fundamental building blocks*? To address this supporting question, an investigation is conducted to understand the techniques that could be used to uniformly develop intelligent semantic services based on the identified fundamental building blocks.*
SQ3: What are the requirements for designing and developing a unified service creation framework, in order to simplify and speedup the process of engineering *IsS*? *To address this supporting question, the objective is to define the requirements for*

[2]***IsS*** is used in this thesis interchangeably to refer to a single intelligent semantic service and plural intelligent semantic services.
[3] **SQ** stands for supporting question.

designing and developing a framework for the purposes of simplifying and accelerating the process of engineering intelligent semantic services.

SQ4: How could the specified service creation framework be implemented in a unified and scalable environment? *In this supporting question, the objective is to practically demonstrate how the proposed conceptual service creation framework could be implemented in a unified and scalable environment.*

SQ5: How can the overall proposed solution be evaluated? *The objective of this supporting question is to apply different research techniques to evaluate the extent to which the main research question has been addressed through the development and implementation of the service creation framework.*

1.4. RESEARCH OBJECTIVES

In order to extensively address the main research question and the supporting questions, the following objectives are identified. The main objective is the investigation and the proposition of a service creation framework that could serve as a blueprint for simplifying and accelerating the process of engineering intelligent semantic services. The following list enumerates additional sub-objectives:

- Formulate an elaborative definition for the term *intelligent semantic service* (*IsS*).
- Identify and characterize a set of fundamental building blocks that make up an *IsS.*
- Devise an appropriate service engineering methodology for developing intelligent semantic services.
- Specify the requirements for the envisaged service creation framework.
- Implement the proposed framework by developing, and/or re-using software artefacts that could contribute to simplifying and accelerating the process of engineering *IsS.*
- Evaluate the proposed and implemented service creation framework against the design requirements, related solutions, appropriate use case scenarios, and based on performance and scalability.

1.5. BENEFITS OF THE STUDY

The main contributions emanating from this study are divided into primary and secondary contributions as briefly explained below.

1.5.1. PRIMARY RESEARCH CONTRIBUTIONS

- Distinct fundamental building blocks that make up an intelligent semantic service. The fundamental building blocks are covered in Chapter 5.
- A service engineering methodology that supports the use of multiple service architectural style and semantic description languages to engineer intelligent semantic services. The proposed methodology is discussed in Chapter 6.
- A multi-layered *service creation framework* to simplify and speedup the process of engineering intelligent semantic services. The service creation framework is proposed and discussed in Chapter 6.
- Advance state of the art of the service engineering domain with innovative service creation frameworks, methods, and tools for constructing intelligent semantic services.

1.5.2. SECONDARY RESEARCH CONTRIBUTIONS

This study also indirectly contributes toward the:

- Promotion and uptake of semantic services and applications.
- Re-usability and interoperability of semantic descriptions during service development.
- Minimization of the time and costs required for engineering intelligent semantic services.
- Provision of a suitable test environment for intelligent semantic services and related applications.

1.6. RESEARCH METHODOLOGY

In order to contextualize this research study and the proposed solution, an extensive literature review was conducted. The literature review was conducted on specific concepts related to this study, such as *Web Services, Semantic Web, Semantic Web Services, ontologies, service engineering, and intelligent agents*. The reviewed literature and related work is presented in Chapter 2 – Chapter 4.

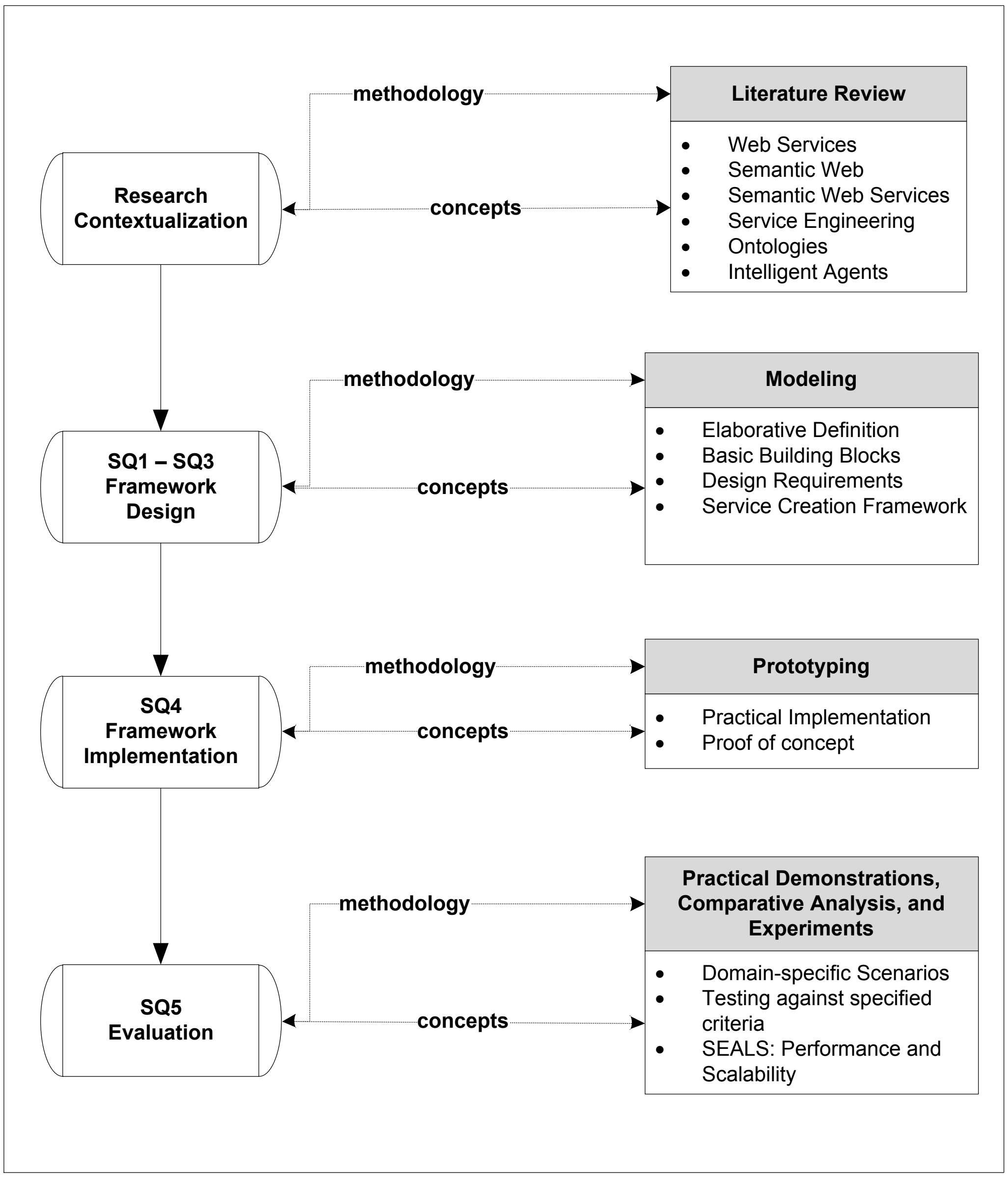


Figure 1.3: Mapping Research Questions to Research Methodologies

As depicted in Figure 1.3, the primary research methodology adopted for this study is *modelling* (Jordaan & Lategan, 2010), where a service creation framework is proposed and implemented. In this context, modelling refers to the creation of a model or framework that captures the components (Olivier, 2006) that are essential in simplifying and accelerating the process of building semantic services. This approach is employed to address SQ1 – SQ3. The modelling method is preferred, as it has been found to be appropriate when capturing or representing the essential components of a system or process (Olivier, 2006:45), particularly for complex systems or processes.

SQ4 deals with the practical implementation of the modelled service creation framework. Thus, for proof of concept purposes, SQ4 is addressed using the *prototyping approach*. According to Olivier (2006:9), prototypes are used to show that new models are plausible and that these could be implemented in practice. Moreover, prototypes are also useful for experimental purposes.

As illustrated in Figure 1.3, SQ5 is addressed by employing three different research evaluation techniques. These include *practical demonstrations*, *comparative analysis*, and controlled *laboratory experiments* - using the SEALS methodology (Wrigley *et al.*, 2011).

Practical demonstrations using domain-specific use case scenarios are conducted to assess the functionality and utility of our service creation framework. With regard to comparative analysis, the proposed framework is theoretically compared against other existing solutions using a comprehensive list of design principles formulated based on SQ3.

According to Hofstee (2006:128), experiments are conducted to “observe the effect of a given intervention”. For the purpose of this thesis, controlled laboratory experiments are also conducted. This is done to gain deeper insight into the service creation framework, and to note the effects (i.e. performance, and scalability) of the framework when engineering intelligent semantic services.

The SEALS[4] methodology is adopted and used specifically for evaluating the performance and scalability of the proposed and implemented service creation framework. The methodology is chosen specifically because it is one of the few readily available, comprehensive, and appropriate approaches for adequately evaluating semantic technologies. Furthermore, the methodology has been employed in evaluating a number of prominent semantic and ontology development tools, such as Protégé[5] (García-Castro *et al.*, 2011).

The SEALS methodology was developed in the European Union (EU) seventh framework programme project called SEALS (Semantic Evaluation at Large Scale) with the purpose of creating a *"lasting infrastructure for evaluating semantic technologies"* (García-Castro *et al.*, 2011). It focuses on evaluating semantic technologies automatically and interactively.

The methodology (i.e. SEALS) considers different criteria for evaluating semantic service technologies (García-Castro *et al.*, 2011). These are briefly explained as follows:

- *Performance* - This refers to the performance of specific activities facilitated by the semantic technology, such as service discovery (e.g. how long does it take to automatically discover services?).
- *Scalability* - This refers to the ability of the semantic technology to perform specific activities involving an increasing number of requirements (e.g. create domain ontologies).
- *Correctness* - This refers to the ability of the semantic technology to respond appropriately and correctly to different requests based on available domain ontologies and semantic descriptions.
- *Conformance* - This refers to the extent to which the semantic technology conforms to the features of the SWS architecture (SWSA) (Burstein *et al.*, 2005).

[4] More information on the SEALS methodology can be found at: http://www.seals-project.eu
[5] http://protege.stanford.edu/

- *Usability* - This deals with the subjective user-friendliness of specific semantic technologies. The SEALS methodology suggests feedback forms as one possible option for measuring the usability of different SWS technologies.

1.7. RESEARCH SCOPE AND LIMITATIONS

The focus of this study is mainly on the simplification and acceleration of the process of engineering intelligent semantic services. An elaborative definition of the term *intelligent semantic service* is detailed in Chapter 5. In addition, the critical discussion in this thesis deals with the concepts of service engineering (SE), Web Services (WS), and Semantic Web services (SWS), ontologies, as well as intelligent agents (IA).

The prototype (i.e. service creation platform) developed for this thesis is intended to demonstrate the proof-of-concept implementation, rather than the actual realization of a fully-fledged unified service creation environment for constructing intelligent semantic services.

It should be noted that other concepts, which also form part of service engineering, such as service discovery, selection, composition, orchestration, and choreography are beyond the scope of this thesis. Nevertheless, occasional reference is made to these concepts in the subsequent chapters, for clarifying core issues related to the proposed solution.

1.8. PUBLICATIONS

In this section, we highlight the key publications that emanated from this research study.

- MTSWENI, J., BIERMANN, E. & NGASSAM, E.K. 2009. Towards flexible engineering of intelligent semantic-based services: building blocks and methodology. Poster presented at the SAICSIT Conference, October 2009

- MTSWENI, J., BIERMANN, E. & NGASSAM, E.K. 2009. Towards the engineering of intelligent semantic-based services: building blocks and methodology. SAICSIT M&D Symposium. September 2009
- MTSWENI, J., BIERMANN, E. & PRETORIUS, L. 2010. iSemServ: Towards the Engineering of Intelligent Semantic-based Services*, ICWE Workshops 2010, 550-559, Vienna, Austria*
- MTSWENI, J., BIERMANN, E. & PRETORIUS, L. 2010. Toward a service creation framework: a case of intelligent semantic services. *In:* Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists. Bela Bela, South Africa.
- MTSWENI, J., BIERMANN, E. & PRETORIUS, L. 2012. *i*SemServ: Facilitating the Implementation of Intelligent Semantic Services. Accepted for presentation at the 9th International Network Conference 2012, Port Elizabeth, South Africa 11-12 July 2012.

1.9. THESIS OUTLINE

The remaining chapters of this research study are structured as follows: *Chapter 2* gives background information with regard to service-oriented computing (SOC), with the specific focus on WS and SW. In *Chapter 3*, service-oriented engineering techniques are discussed, and their relevance in addressing some of the identified challenges will be highlighted. Prominent semantic models and some of the existing supporting tools are discussed in *Chapter 4*. The proposed solution is formulated, by giving a comprehensive definition of the term intelligent semantic services and formulating the fundamental building blocks that make up intelligent semantic services in *Chapter 5*.

The service creation framework as one possible solution for the challenges identified in Section 1.2 is proposed and described together with its salient modules in *Chapter 6*; and the description is preceded by detailing the service creation framework design requirements. *Chapter 7* discusses the proof-of-concept implementation of the suggested service creation framework, including the technologies essential to the overall implementation. The implemented service creation framework is evaluated and the results are discussed in *Chapter 8.* The thesis is concluded in *Chapter 9,* by discussing the research contributions, and some of the remaining challenges not yet addressed by the suggested service creation framework. These issues could form the basis for further research.

CHAPTER 2: Service-Oriented Computing

This chapter presents background information related to the concepts of service-oriented computing, which is the basis for semantic services. The focus is on the components of Web services, Semantic Web, and semantic Web services. RPC-based and RESTful services as the main architectural styles for distributed Web services are also discussed, including some of their distinct differences.

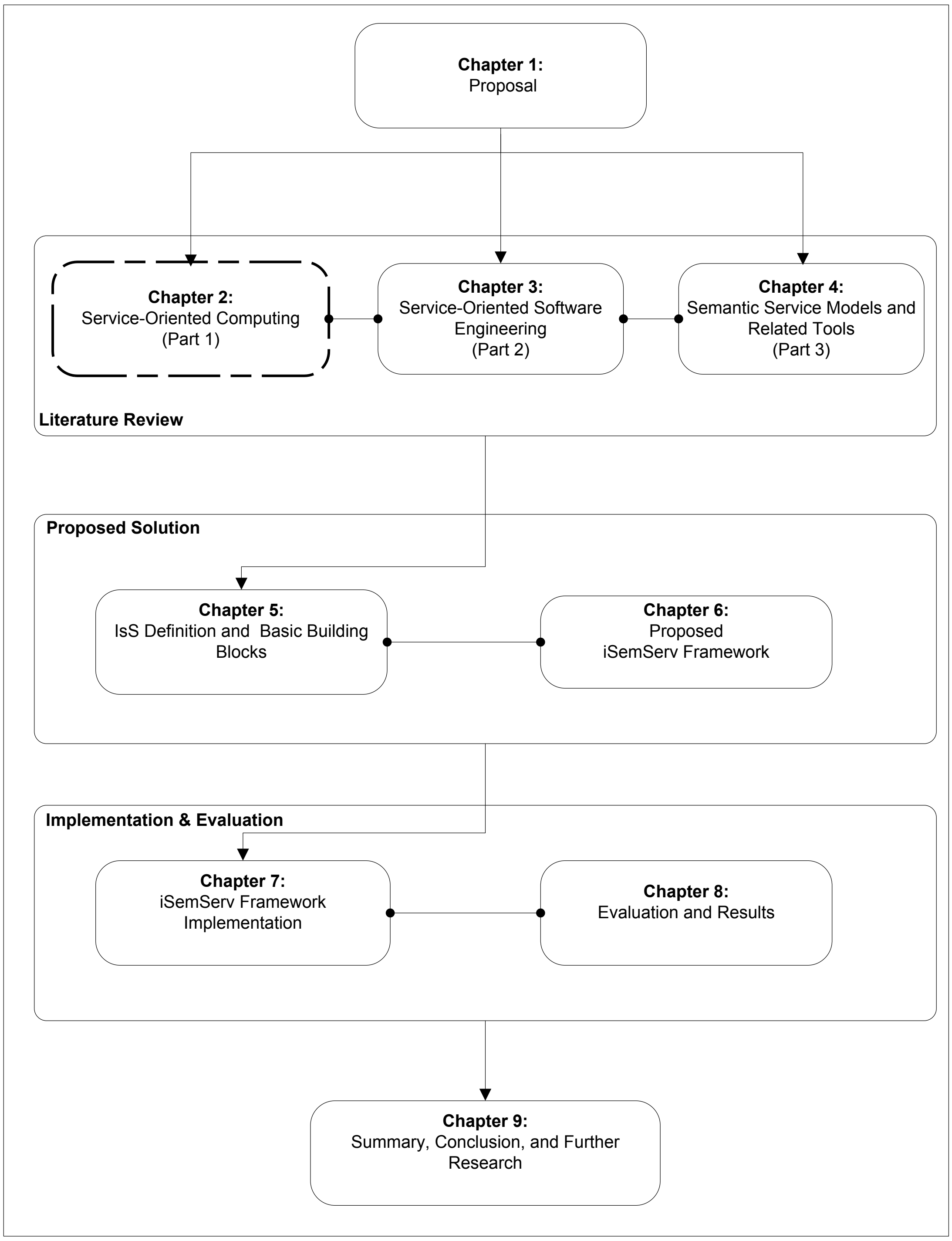

Chapter 1:
Proposal
Chapter 2:
Service-Oriented Computing
(Part 1)
Chapter 3:
Service-Oriented Software Engineering
(Part 2)
Chapter 4:
Semantic Service Models and Related Tools
(Part 3)
Literature Review
Proposed Solution
Chapter 5:
IsS Definition and Basic Building Blocks
Chapter 6:
Proposed iSemServ Framework
Implementation & Evaluation
Chapter 7:
iSemServ Framework Implementation
Chapter 8:
Evaluation and Results
Chapter 9:
Summary, Conclusion, and Further Research

Figure 2.1: Overall Thesis Structure

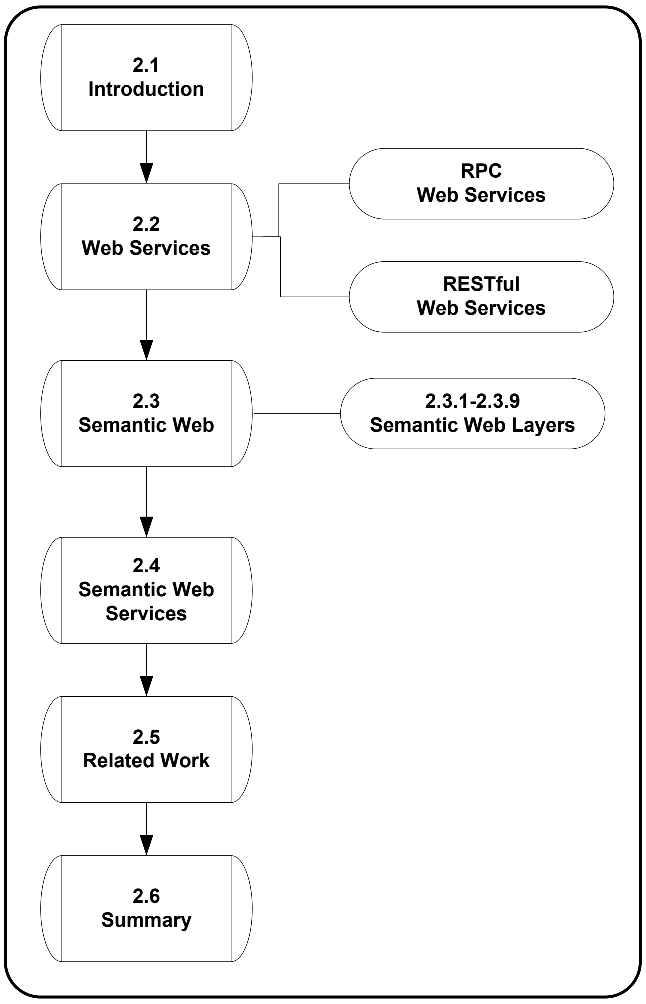


Figure 2.2: Chapter 2 Layout

2.1. INTRODUCTION

Several organizations in the private and the public sector are presently developing and providing some form of Web-based services. For instance, various service providers now deliver services, such as e-learning, e-banking, e-commerce; and more recently, m-banking and m-commerce. As a result, within the current World Wide Web (WWW), different types of services exist; and other types are emerging, to improve on the capabilities of the existing services.

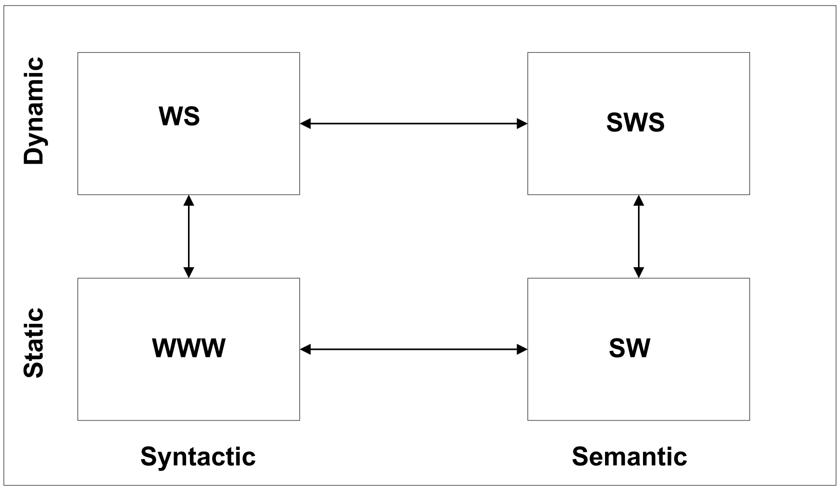


Figure 2.3: Evolution of the Web

Figure 2.3 illustrates how the WWW has evolved since its inception. In essence, the Web is moving towards a collection of semantic services and data supported by the Semantic Web (SW). As discussed in Chapter 1, SW is the extension of the current WWW, with the main objective of promoting a Web that is understandable and processable by both humans and machines (Berners-Lee, Hendler & Lassila, 2001).

In the SW, information and data on the Web are linked by using *ontologies* (Gruber, 1993), in order to enable automatic discovery of, and reasoning over Web content (Bensaber & Malki, 2008). Since the WWW and SW deal mainly with content (i.e. static data), Web Services (WS) augment the Web with integration and computation capabilities, whilst Semantic Web Services (SWS) focus on automating the core tasks of WS, such as discovery and composition, by minimizing user-intervention (Corcho *et al.*, 2003).

In this chapter, the current state-of-the-art pertaining to Web services and related concepts, such as SW, SWS, and ontologies is reviewed and discussed within the context of this study. In addition, related research efforts within the domain of semantic service engineering are also highlighted.

2.2. WEB SERVICES

The main goal of Web Services is to facilitate worldwide accessibility of business functionalities (Filho & Ferreira, 2009; Shen *et al.*, 2005) on the Web. This suggests that WS are mainly created and utilized, in order to perform business-specific tasks that could benefit both the providers and consumers of services (Gottschalk *et al.*, 2002:170; Hassanzadeh, Namdarian & Elahi, 2011).

WS are designed and developed to provide an environment for enabling interoperability between different software applications that are running on different platforms (Booth *et al.*, 2004). WS, as modular applications, can be advertised, discovered and executed across the Web (Kanellopoulos & Kotsiantis, 2006). Furthermore, WS are distinct from general services that can also be accessed over the network. The difference is: Web services have standardized and uniform interfaces that describe all the operations necessary for interaction with other systems (Alonso *et al.*, 2004). Hence, a Web page that provides some business functionalities is not a Web service; but it is rather an e-service.

The concept of WS revolves around three role-players. These role-players are: (1) The service provider; (2) the service consumer; and (3) the service registry (Kanellopoulos & Kotsiantis, 2006). The provider is responsible for defining, developing, and publishing Web services. The requester is primarily the consumer of the advertised and discovered services. For plausible discovery, invocation, and execution by consumers, WS need to be published into service registries (Gottschalk *et al.*, 2002:172).

In the following subsections, we shall discuss the common types of Web services that service developers would generally develop and publish into service registries.

2.2.1. RPC WEB SERVICES

Figure 2.4 depicts the core elements that make up the Remote Procedure Call (RPC) Web Services stack. RPC WS are one possible implementation of the Service-Oriented Architecture (SOA) (Hassanzadeh, Namdarian & Elahi, 2011); and they have enjoyed considerable acceptance within the service engineering domain.

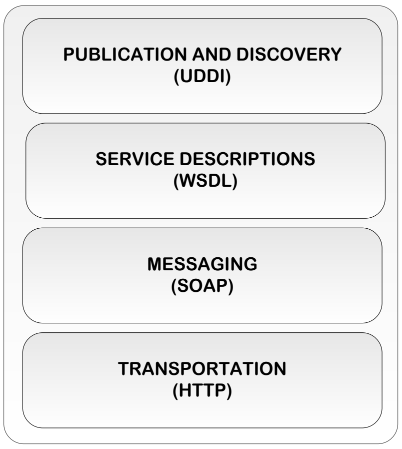


Figure 2.4: Generic Web Services Stack

The RPC WS are well-grounded on standards built on top of existing Internet protocols. Thus, the transportation component in the stack, as illustrated in Figure 2.4, is one of the core layers of RPC WS. The protocols in the transportation layer facilitate the publication, discovery, and invocation of RPC WS over the Web (Shen *et al.*, 2005).

In the messaging layer, the Simple Object Access Protocol (SOAP) is an Extensible Mark-up Language (XML) (W3C, 2005) messaging protocol that handles message exchanges (e.g. input and output messages) between WS and consumers. SOAP messages are normally wrapped around the HTTP protocol; and these could be utilized over various Internet communication protocols that are compatible with HTTP (Keidl & Kemper, 2004).

The Web Services Description Language (WSDL[6]) resides in the service descriptions layer. It exploits the XML language with standardized schemas to

[6] Extensive technical details on WSDL can be found at: http://www.w3.org/TR/wsdl

syntactically define and describe WS capabilities (Kelly, Coddington & Wendelborn, 2006). However, services described with WSDL lack semantic descriptions, which are essential for achieving semantic services.

As depicted in Figure 2.4, the publication and description functions of WS are supposed to be handled by the Universal Description Discovery and Integration (UDDI) standard protocol. According to Garcia-Sanchez *et al*. (2009), UDDI provides all the necessary mechanisms for service providers to advertise WS, and similarly for service consumers to search and locate information on the available services. Nevertheless, functional public UDDI registries are infrequently available today, since organizations, such as IBM and Microsoft, have since discontinued the hosting[7] of public UDDI registries. As a result, other non-UDDI service registries, such as Seekda[8] have emerged.

As may be noted from the description above, the lifecycle of WS is not cumbersome. For instance, a service provider describes a WS using WSDL. The publication of a defined WS is done within a UDDI registry. A consumer, who would like to access a service, firstly needs to discover such a service from the selected service registry. If an appropriate WS is discovered, it can then be invoked and executed, according to the input and output specifications.

The most important element of RPC WS is the description (i.e. WSDL). Descriptions enable services to be discovered, and subsequently, invoked.

2.2.2. RESTFUL WEB SERVICES

Apart from RPC WS, which are often referred to as SOAP-based WS, there are other types of WS emerging, such as RESTful Web services. Representational State Transfer (REST) is an architectural approach that leverages the HTTP-based protocol; and it is considered to be simpler and more lightweight, compared with the well-established architectural styles, such as RPC (Fielding, 2000; Filho & Ferreira, 2009; Pautasso, Zimmermann & Leymann, 2008). It was originally proposed for

[7] The news regarding the discontinuation of the UDDI registry hosted by IBM *et al*., can be found here: http://soa.sys-con.com/node/164624
[8] See: http://webservices.seekda.com

large hypermedia systems, and stresses the scalability, generality, and independency of resources on the Web (Fielding, 2000).

Web services that are considered *RESTful* need to conform to REST specifications. These include:

- **Identification of *resources***: a RESTful service needs to be identified as a resource. A resource is any concept that could be represented or named in a system (e.g. an image).

- ***Representations***: RESTful services need to specify their representations that could be manipulated by service consumers. In this regard, representations are metadata about a resource (Fielding, 2000). RESTful services support various representations, such as JSON, HTML, and XML: unlike RPC-based services; which only support XML.

- ***Uniform identifiers:*** any RESTful service needs to be identified with a Uniform Resource Identifier (URI)[9]. The uniform identifier is used as both the name and locator of the RESTful service during discovery and invocation.

- ***Unified interfaces***: RESTful services must specify standard operations that could be performed on the resources. These are mainly the common HTTP methods, such as GET, POST, PUT, and DELETE, as illustrated in Table 2.1.

Table 2.1: HTTP Methods

HTTP METHODS	CRUD OPERATIONS
@POST	Create
@GET	Read
@PUT	Update
@DELETE	Delete

[9] See: http://www.w3.org/TR/uri-clarification/

- ***Execution scope***: RESTful resources need to also define the execution scope of services, such as the aspects of a resource that need to be affected; for example: input and output parameters (Filho & Ferreira, 2009).

Fielding (2000) further describes the principles that apply to the REST architectural style. Some of these are:

- ***Stateless****:* This means that each connection to the server by the client includes all the information necessary to fulfil the request.
- ***Cacheable****:* Responses from a REST-enabled server can be implicitly or explicitly labelled cacheable (i.e. responses can be stored by clients for re-use) or non-cacheable.
- ***Addressable****:* Every REST-compliant resource needs to have one or more addressable Uniform Resource Identifiers.

RESTful Web services, a term coined by Richardson and Ruby (2007), conform to the principles and specifications prescribed by the REST architectural style. They do not use the SOAP protocol or the architecture used by RPC WS. Similar to RPC WS, RESTful services need to be described, so as to be discovered by potential consumers. As indicated in Section 2.2, WS employs the popular WSDL for syntactic descriptions. On the contrary, RESTful services do not boast of a standard syntactic description language as yet, although, WSDL2.0 (Chinnici *et al.*, 2007) does accommodate the description of RESTful services.

Nevertheless, an XML-based description language, referred to as Web Application Description Language (WADL[10]) is beginning to be used favourably by RESTful services developers. It is provides machine process-able descriptions for web-based services and applications (Hadley, 2009). In addition, WADL is also considered light-weight compared to WSDL, due to its reliance on open protocols, such as HTTP.

With regard to publication and discovery, RESTful services do not prescribe any new standard for service registries.

[10] Extensive technical specifications for WADL are provided here: http://www.w3.org/Submission/wadl/

2.3. SEMANTIC WEB

The WWW is made up of large amounts of data and information, which are presented in a format that can mostly be processed and understood by humans (Cowles, 2005). Hence, the Semantic Web (SW) or Web 3.0, as referred to by Cardoso (2007b:84), is evolving to provide well-defined meaning to information and services on the Web, so that humans and computers can work better together.

As envisaged by Berners-Lee (2003), in theory, SW aims to:

- Enable machine-interpretable Web resources;
- Augment Web resources with concepts and relations;
- Bridge the gap with regard to data integration across heterogeneous applications and organizations;
- Automate a variety of Web-based tasks, such as search, discovery, composition, invocation, and execution of services. As a result, it should be possible to reduce human intervention in a number of Web-based tasks.

The concept of SW is generally made possible by embedding machine-interpretable content into Web resources, such as documents and services (Oberle *et al.*, 2005:328). Accordingly, machine-understandable content is achieved through the use of ontologies, which are an integral part of the SW, as they facilitate the representation of knowledge on the Web (Tho, Fong & Hui, 2007).

Cowles (2005) explains the overall concept of SW as follows: “As the Semantic Web gains momentum, an increased number of information resources will be just as useful to software agents (i.e. machines) as to humans”. Hence, the SW is intended to ensure that computers are able to accurately process and understand information on the Web without any user intervention.

Within the SW research domain, there are immediate efforts toward the formalization of standards and development of semantic technologies that are intended to enable the overall vision of SW (Joo, 2011; Oberle *et al.*, 2005:328). These standards and technologies are envisaged to advance, amongst others, information searching, data integration, and the automation of Web-related tasks (Koivunen & Miller, 2002).

According to Cabral *et al.* (2004), SW enabling technologies and standards are structured into a set of layers (Berners-Lee, 2000), as depicted in Figure 2.5. A combination of these layers is referred to as an overall Semantic Web Architecture (Gerber, Barnard & van der Merwe, 2007). However, there are a few versions of the SW architecture that have been proposed, and improved, over the recent past (Al-Feel, Koutb & Suoror, 2009; Berners-Lee, 2000, 2003, 2005, 2006). In this study, we only focus on the generic SW technologies and standards, as found in most of these different versions of the SW architecture.

Figure 2.5 depicts the current SW architecture version (Berners-Lee, 2006). According to Al-Feel, Koutb and Suoror (2009), and Horrocks *et al.* (2005), all the layers depend on each other, and each layer uses the features of the layer below, and extends the capabilities of the layer above. As alluded to by Gerber, Barnard, and van der Merwe (2007), the current SW architecture presents both the functional and technological aspects. However, this is not consistent in all the layers.

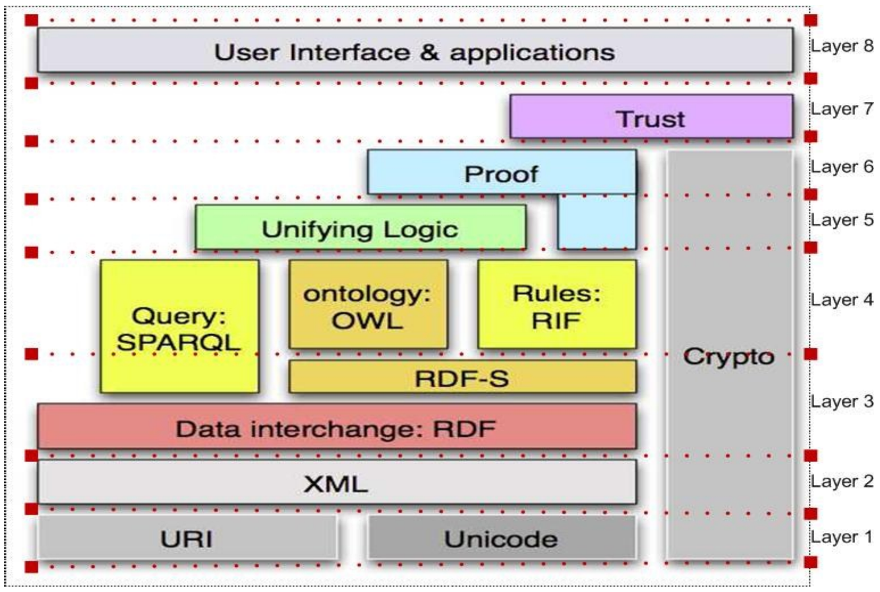


Figure 2.5: Current Semantic Web Architecture

In the following subsections, the discussion will generally focus on the functionalities that each layer supports, especially functionalities that are the keys to the proposed study.

Layer 1: URI and Unicode

Layer 1,as a foundation layer, is responsible for unambiguously identifying and representing resources on the Web, using a compact sequence of characters (Al-Feel, Koutb & Suoror, 2009:808; Berners-Lee *et al.*, 2005). Secondly, this layer is responsible for encoding characters from any written language, enabling users and machines to use any language for data representation on the Web. *URI* and *Unicode* are the common technologies available today to implement the functionalities of the first layer (Al-Feel, Koutb & Suoror, 2009:808).

According to Berners-Lee *et al.* (2005), URI supports the identification and representation of resources on the Web; whilst the Unicode standard identifies and encodes any international characters linked to different Web resources (Gerber, Barnard & van der Merwe, 2007).

Layer 2: XML

This layer supports the storage and the exchange of semantic data between machines and users on the SW (Al-Feel, Koutb & Suoror, 2009:808) through the utilization of standard technologies, such as XML. The XML standard promotes common syntax usage in the SW (Obitko, 2007), thus boosting interoperability between different systems and applications. Aziz *et al.*, (2004) further explain that in this layer, XML facilitates the process of defining semantic contents and rules through the use of XML namespaces and schemas, which are responsible for primarily describing the structure of an XML-based document.

Layer 3: RDF and RDF-S

Resource Description Framework (RDF), as an XML-based standard, simply "describes resources with URI on the Web" (Cabral *et al.*, 2004). This is accomplished by linking Web resources with well-defined semantics, (i.e. RDF data) interpretable and processable by machines (Lassila *et al.*, 2000:67). In essence, RDF augments layer 2 (i.e. XML standard) with semantics by “allowing the description and representation of resources through properties”.

RDF-Schema (RDF-S) is a basic type of system that enables the provisioning of metadata for processing and interpreting the RDF data (Cabral *et al.*, 2004; Gerber, Barnard & van der Merwe, 2007).

Nevertheless, RDF has some limitations, since it does not provide richer semantics and support for describing cardinality constraints, which are some of the important aspect of ontologies (Horrocks, 2008).

Layer 4: Query, Ontology, and Rules

The *Query* function is incorporated in the *fourth layer* for querying and retrieving RDF data, RDF metadata, and ontologies, so that they can be interpreted and processed by machines. Simple Protocol and RDF Query Language (SPARQL) is the recommended SQL-like language; and a protocol for querying and accessing relevant semantic data and ontologies (Obitko, 2007).

Ontologies are core to the overall SW architecture. They are basically used for formally representing some knowledge within a specific domain. The term ontology is commonly defined as *“a formal specification of a shared conceptualization”* (Gruber, 1993).

The Web Ontology Language (OWL) is one of the languages available for implementing the ontological functions of layer 4 (Obitko, 2007), and it provides richer semantics as compared with the RDF in layer 3. In fact, OWL is one of the common languages used today for generating Web ontologies (Cardoso, 2007b:85). Computers use the defined ontologies to automatically interpret and process Web information and services with limited human assistance (Hernandez, 2007).

Rules also plays a major role in the SW (Eiter *et al.*, 2008). The rules component in this layer is aimed at easing the automatic reasoning and transformation of knowledge on the Web by machines (Paschke & Bichler, 2008). Al-Feel, Koutb and Suoror (2009:809) affirm that rules in the SW are meant to simplify querying, reasoning, and the filtering of semantic data. Rules Interchange Format (RIF) and Rule Mark-up Language (RuleML) are two of the promising languages for realizing the rules component of this layer.

Layer 5: Unifying Logic

The *unifying logic layer* supports layer 4, in particular ontologies and rules with logical deductive reasoning and dynamic inference of the semantic data (Aziz *et al.*, 2004:368; Hyvonen, 2002:16). This enables machines to automatically deduce the meaning and purpose of the knowledge and the rules defined in layer 4.

Layer 6: Proof

The *proof layer* deals with validating and confirming the knowledge produced for the Semantic Web using some ontological language such as OWL. This layer attempts to assure SW users (e.g. agents) that the deduced knowledge, as developed using ontologies, is correct (Al-Feel, Koutb & Suoror, 2009:809; Hyvonen, 2002:16). Currently, a language called Proof Mark-up Language (PML) (Da Silva, McGuinness & Fikes, 2006) is one of the implementation examples of the proof layer.

Layer 1-6: Cryptography

This layer spans across layer 1 to layer 6; and it is responsible for applying the overall security to the semantic data. It applies W3C recommended technologies, such as encryption, decryption, and digital signatures – to ensure that the SW resources are secured and trusted (Al-Feel, Koutb & Suoror, 2009:809).

Layer 7: Trust

The *trust layer* is aimed at ensuring the trustworthiness of the domain knowledge (i.e. ontologies) made available in layer 4.

Layer 8: User Interface and applications

This top-most layer of the SW architecture represents the platform where Web applications can be SW-enabled. This layer enables users and agents to use SW-enabled applications. There are a number of applications that could form part of this layer, such as *information* and *search retrieval, e-marketplaces, knowledge management,* and *intelligent e-commerce Web applications* (Hyvonen, 2002:17).

The background details discussed in this section serve as an input to the next section, where this research study is grounded.

2.4. SEMANTIC WEB SERVICES

Although Web services focus on the accessibility of business functionality over the Web, and promote interoperability amongst heterogeneous business applications, various challenges are still being experienced in this field, such as the lack of semantic descriptions that could enable WS to be fully machine-processable and interpretable. The emergence of Semantic Web Services (SWS) has been purposed as a possible solution for the challenges of WS. Dimitrov *et al.,* (2007) assert that WS without semantic descriptions are less dynamic; hence, the move towards SWS is an essential one (Janev & Vranes, 2010) in order to achieve automation in Web services.

In essence, SWS is the confluence of WS and the SW, to create services that are capable of activities, such as automatic discovery, composition, and execution (Agre *et al.*, 2007). Within the SWS sphere, several standards and languages are evolving on how to develop semantic services. These include standards and languages, such as Web Ontology Language for Services (OWL-S), Web Services Modelling Ontology (WSMO), based on Web Service Modelling Language (WSML), Semantic Web Services Language (SWSL), and Web Service Semantics (WSDL-S) (Akkiraju *et al.*, 2005; Battle *et al.*, 2005; García-Sanchez, 2007; Roman *et al.*, 2006; Smith, Welty & McGuinness, 2004).

WSMO is one of the approaches that is emerging to facilitate ontology development, particularly when describing various aspects of SWS such as service capabilities (Acuna & Marcos, 2006:33). OWL-S is a pure service ontology language based on OWL. OWL-S is described as one of the major SWS description languages (Bensaber & Malki, 2008). These emerging standards and languages mainly focus on the description, publication, discovery, selection, composition, and invocation of services (Cabral *et al.*, 2004). In Chapter 4, some of these common SWS standards, languages, and technologies will be discussed further.

According to Lu, Zhang and Ruan (2007), SWS still faces some challenges, such as personalization, customization and engineering (i.e. design and development). This is evidenced from the lack of adoption and development tools (Bensaber & Malki, 2008; Bouchiha & Malki, 2010) meant to ease the process of building of such services.

It is also important to highlight that the complexity of SWS languages and standards is generally not hidden from the user, thus making it a challenge for semantic services to be widely adopted and exploited by service developers. Thus, the main objective of this study is on simplifying and accelerating the process of building intelligent semantic services.

The following section elaborates on some of the recent research efforts that have been conducted within the scope of semantic service engineering and semantic technologies.

2.5. RELATED WORK

In this section, we highlight some of the common related studies that attempt to address some of the challenges of engineering semantic services. The focus is exclusively on the studies that provide an end-to-end development lifecycle of semantic services: from service design to service deployment.

The domain of semantic services is still in its infancy stage, and the research activities are mainly on the different concepts of SWS, such as automatic service discovery, composition, and execution. In terms of semantic service engineering, there have been a limited number of studies conducted, since the emergence of the concept of SWS (Agre *et al.*, 2007; Anaby-Tavor. *et al.*, 2008; McIlraith, Son & Zeng, 2001).

Stollberg *et al.,* (2004:5) proposed a Semantic Web Fred (SWF) mediation platform for building agent-based applications, based on different use case scenarios. The focus in SWF is on creating an agent that represents an e-service. In this platform, SWS are integrated from external sources, rather than being developed within the platform. Furthermore, this platform is tightly coupled to one specific ontology language (i.e. WSMO).

One approach that attempts to alleviate the problems of SWF is a framework by Garcia-Sanchez *et al.,* (2009). The framework, called SEMMAS (SEMantic web services Multi-Agent System), is independent of the domain and application to which it is applied. It is made up of four layers that cover various aspects, such as the business logic, SWS, agents, and applications. SEMMAS does not prescribe a particular ontology language. The main challenge of SEMMAS is that it relies heavily on external WS. Moreover, the creation of domain and application ontologies is manually achieved. Issues of complexity hiding and simple engineering of intelligent semantic services are not addressed.

ODE-SWS, a SWS development environment of Corcho *et al.,* (2003) focuses on developing SWS in a language-independent approach. Various SWS languages can be used within this platform. The framework is integrated within WebODE, an ontology engineering workbench, responsible for exporting provided ontologies into other ontology languages (WebODE, 2003). The main limitation of ODE-SWS is that complete service automation, as prescribed in SWS, is not addressed at all.

One of the other frameworks that claims to be the first in SWS engineering is called INFRAWEBS (Agre *et al.*, 2007). It focuses on constructing semantic descriptions for existing and new WS; and it enables the integration of disparate components (e.g. WS and SWS). INFRAWEBS is made up of different units (i.e. SWS creation, monitoring, selection, discovery, composition, and conversion); these are paramount in the actual development and implementation of semantic services. INFRAWEBS suffers from the same limitations as SWF; that is, it is bound to a specific ontology language. Furthermore, it assumes that ontologies are already defined, and can therefore be re-used.

A framework called Internet Reasoning Service (IRS-III) is a comprehensive framework and a platform for creating WSMO-based SWS (Cabral, 2006; Domingue *et al.*, 2008). IRS-III is promoted as a development framework for SWS. According to Domingue *et al*. (2008:110), the main goal of IRS-III is to support capability-based discovery and invocation of semantic services. Web applications can be built and executed within this platform, using the IRS-III Browser (Cabral, 2006). Its other main role is to mediate between service providers and service consumers using ontologies to further enhance the interoperability and collaboration (Domingue *et al.*, 2008:110). However, IRS-III does not provide an environment, where new semantic services can be engineered, based on the users' requirements.

The IRS-III framework, which is made up of an *IRS-III Server, Publisher, and Client* does, however, provide support for creating semantic applications out of existing SWS (Cabral, 2006) and out of existing Java and Lisp code (Domingue *et al.*, 2004). Moreover, ontologies can be generated using Operational Conceptual Modelling Language (OCML) and WSMO.

Lastly, one of the research endeavours that is also related to this study, is that of Srinivasan, Paolucci, and Sycara (2005). The authors proposed and realized a practical integrated development environment (IDE) called OWL-S IDE, formally known as CMU[11]'s OWL-S Development Environment (CODE) for developing, deploying, and consuming semantic-based services.

OWL-S IDE adopts and extends existing WS tools such as OWL-S editor and WSDL2OWL-S converter in order to support developers in the process of developing, deploying, and consuming semantic services (Srinivasan, Paolucci & Sycara, 2006). It is embedded within the Eclipse[12] environment, and is purely based on Java and OWL-S. It follows a multi-approach, by applying both code-driven and model-driven methodologies in delivering semantic services.

The OWL-S IDE platform also supports various SWS activities, such as discovery, invocation, and execution. However, it does not cater for interoperability and high dynamism (i.e. automation) as envisaged with the overall vision of SWS.

[11]Carnegie Mellon University

[12] Eclipse is an open source Java IDE that supports languages, such as Java, C/C++, and PHP. For more information visit: http://www.eclipse.org

Table 2.2: Summary of Related Work

RELATED WORK		SWF	INFRAWEBS	OWL-S IDE	IRS-III	SEMMAS	ODE-SWS
Authors		Stollberg et al (2004)	Agre et al. (2007)	Srinivasan, Paolucci, Sycara (2006)	Domingue, et al. (2008)	Garcia-Sanchez et al. (2009)	Corcho et al. (2003)
Programming Language	METHODOLOGY	Java/C++/VB	Java	Java	Java & Lisp	Java	Java
Model-driven		No	No	Yes	No	No	Yes
Code-driven		Yes	Yes	Yes	Yes	No	No
Lifecycle		integrate, publish, discover, invoke, execute	create, compose, discover, select, execute, monitor	develop, describe, publish, discover	broker, create, publish discover, invoke, choreograph, orchestrate, execute	discover, select, compose, invoke, coordinate, negotiate, manage & monitor	design, develop, describe, publish, discover, compose
Architectural Style		SOAP	SOAP	SOAP	SOAP	SOAP	SOAP
Ontology Language		OXML	WSMO	OWL	OCML	OWL-DL	WebODE
Service Description Language		WSDL	WSDL	WSDL	WSDL	WSDL	WSDL
Semantic Description Language		OXML	WSML	OWL-S	OCML	OWL-S	DAML-S
Intelligence		Ontologies Agents	Ontologies	Ontologies	Ontologies	Ontologies Agents	Ontologies

Table 2.2 summarizes the related work reviewed. The related work was reviewed based on the generic features of WS, SWS, and the overall objectives of this study. The features exploited for the review are explained as follows:

- ***Development Methodology:*** This was to determine the methodology followed by the related solutions to address the engineering of semantic services. Two methodologies were identified across these related solutions, that is, *code-driven* and *model-driven*. Code-driven simply means that the solution follows a bottom-up approach, where code is the foundation of every component that forms part of intended semantic services. On the contrary, model-driven engineering follows a top-down approach, where models are the cornerstone for building systems. Programming languages supported by the existing solutions were also identified. It became apparent that Java was the language of choice across all the related solutions.

- ***Life* Cycle:** In this criterion, the goal was to ascertain the phases of semantic service development that each solution supports. It was ascertain that most of the solutions focus on the phases that are beyond the actual development of semantic services, such as discovery, and execution. Thus, the key focus of this study is to address challenges pertaining to the design and development of semantic services.

- **Architectural Style:** An assessment was also made to determine the architectural styles that each of the related solution supports. It was gathered that all of them are inclined to the RPC-based architectural style (e.g. SOAP-based services).

- **Service Description Language:** The related solutions were also reviewed, according to the types of service description languages that they support; and all the solutions support WSDL descriptions.

- **Ontology Language:** All the solutions were also reviewed, based on the ontology languages that they support for defining domain knowledge. In this case, it was clear that different ontology languages are supported by different solutions.

- **Semantic Description Language:** In order to corroborate that different solutions support different ontology languages, a review of semantic description languages supported by related solutions was also conducted. As can be noted in Table 2.2, different solutions were found to be supporting only one semantic description language.

- **Intelligence:** An analysis was conducted to find out how each solution addresses the issue of intelligence[13] in Web services. It was discovered that most solutions rely only on ontologies to achieve intelligence in Web services, whilst some solutions combine ontologies and agents to realize intelligence in Web services.

2.6. SUMMARY

The focus of Chapter 2 has been on the concepts of service-oriented computing (SOC). In this chapter, WS are referred to as the loosely coupled distributed Web-based artefacts that represent the implementation of business services. They are normally accessed using open XML-based standard protocols, such as SOAP. Furthermore, they are grounded in common technologies, such as WSDL for syntactic service descriptions, UDDI for registration, and SOAP for messaging between the service provider and service consumer.

One of the existing drawbacks of WS, is that they are purely described in a syntactic manner; thereby, presenting a challenge when autonomously processed and consumed by software programs. Hence, SWS are emerging to address this particular challenge.

In essence, SWS are merely an extension of WS with the SW technologies. SW, as described in the previous sections, is also an extension of the current Web, where

[13]The term *intelligence* in this study has a similar connotation with terms such as high dynamism, and automation. An elaborated definition of what is meant by intelligence in the context of this study is provided in Chapter 5.

the vision is to enable all the data on the Web to be strategically linked, in order to facilitate the process of interoperability and automation on the Web. The main pillar of SW is ontologies; these provide the possibility of describing data and services on the Web semantically; thereby, contributing towards making it possible for software programs to unambiguously understand the Web and its content.

In this chapter, we have also presented a summary of related work, especially with regard to the solutions that are closely linked to the work proposed in this study. Although SWS is still immature in terms of development platforms and tools, a number of researchers have made some strides in ensuring that SWS should become a reality. Some of the related work that was covered in this chapter, includes the work of Stollberg (2004), Garcia-Sanchez *et al.,* (2009), and Corcho *et al.,* (2003). More importantly, comprehensive development frameworks, such as IRS-III and OWL-S IDE, were also discussed and summarily evaluated, in order to determine their relevance to our proposed work, and their limitations in relation to the objectives of this study.

CHAPTER 3: Service-oriented Software Engineering

This chapter continues with the literature review related to the proposed study. Concepts of service-oriented software engineering are reviewed, and discussed. The core of this chapter is the phases that constitute the service-oriented software engineering lifecycle. In approaching our solution, these phases are essential in bringing forth a framework that will promote simple and rapid engineering of intelligent semantic services.

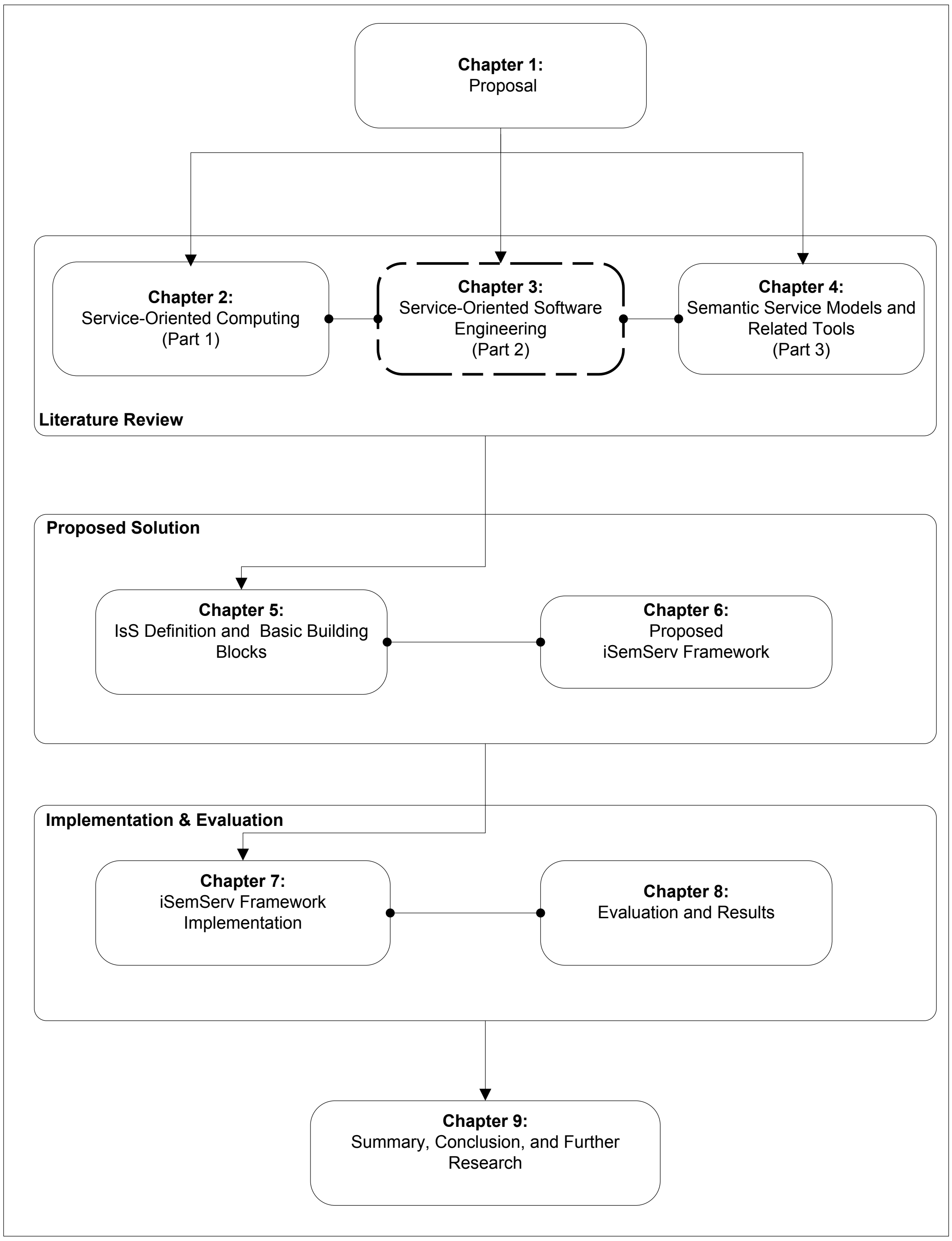

Chapter 1:
Proposal
Chapter 2:
Service-Oriented Computing
(Part 1)
Chapter 3:
Service-Oriented Software Engineering
(Part 2)
Chapter 4:
Semantic Service Models and Related Tools
(Part 3)
Literature Review
Proposed Solution
Chapter 5:
IsS Definition and Basic Building Blocks
Chapter 6:
Proposed iSemServ Framework
Implementation & Evaluation
Chapter 7:
iSemServ Framework Implementation
Chapter 8:
Evaluation and Results
Chapter 9:
Summary, Conclusion, and Further Research

Figure 3.1: Overall Thesis Structure

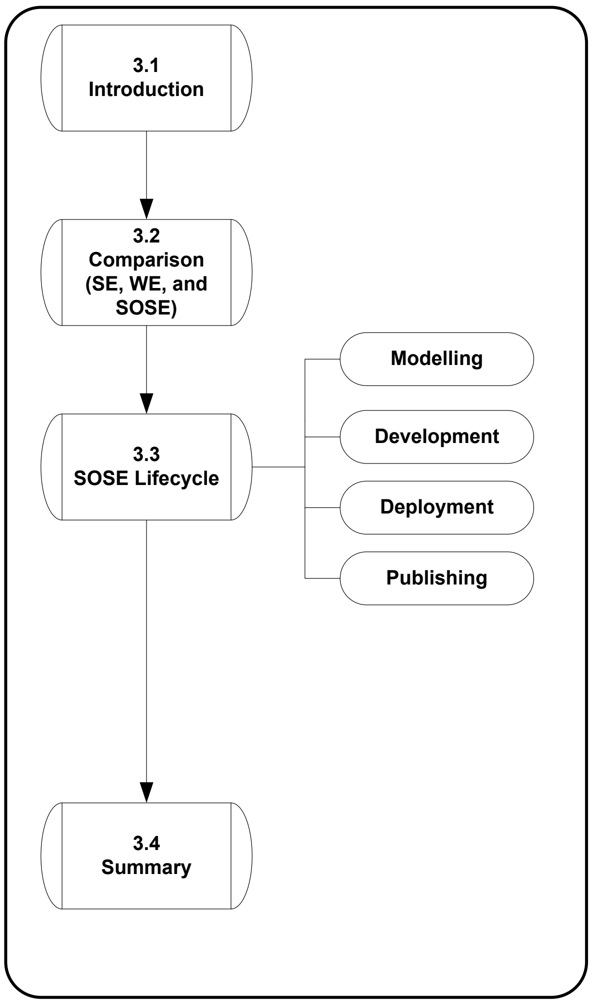


Figure 3.2: Chapter 3 Layout

3.1. INTRODUCTION

Service orientation, as a novel approach to service-based system development, has gained considerable attention in the software development industry over the years (Kontogiannis, Lewis & Smith, 2008; Tsai, 2005). Hence, there has been a paradigm shift, as modern enterprises are slowly moving away from traditional software development to service-oriented system development; where software systems are developed by composing cross-organizational open services (Gu & Lago, 2007; Hassanzadeh, Namdarian & Elahi, 2011).

According to Simula (2007), trends indicate that the life cycle of software applications is becoming relatively shorter than before. Thus, there is a preference for service-oriented system development; which caters for software systems that can be developed and deployed over a short period of time. This shift to service-oriented system development could further be attributed to the growth and development of the World Wide Web (WWW). Web content (i.e. data and services) is produced and delivered on the Web on a daily basis by individuals and businesses, leading to a Web consisting of astronomical amounts of data and services (Sheng *et al.*, 2010:186). Accordingly, this evolution calls for frameworks, easier-to-use methods, and tools that could simplify the process of delivering and consuming Web contents, especially semantically rich services and service-oriented systems.

Service orientation is an approach that advances the development of software applications, by using the concepts of Web services (Kontogiannis *et al.*, 2007; Stojanovic & Dahanayake, 2005:1); which are platform-independent, leading to seamless integration of heterogeneous systems. A number of software development enterprises are adopting and applying Service-Oriented Architecture (SOA), as a preferred method for producing and delivering service-based applications on the Web (Chen, 2008; Kontogiannis *et al.*, 2007).

Some of these enterprises, such as IBM, Oracle, SAP, and others have adopted SOA for the purposes of lowering software production/re-production costs, and at the same time, promoting service re-usability and interoperability (Hassanzadeh, Namdarian & Elahi, 2011; Yu & Ong, 2009).

SOA focuses on three simple roles briefly explained in Chapter 2, Section 2.2.1. These roles are those of a *service provider*, a *service broker*, and a *service consumer,* as depicted in Figure 3.3. Each role player has a set of activities or tasks to perform in Service-Oriented Software Engineering (SOSE). These activities are extensively discussed in Section 3.3. However, it should be noted that in this study the focus is mainly on the engineering phases that are meant to take place in the service providers' environment.

This means that the core focus is on the activities that specifically deal with service production. Other activities that are executed within the service consumer and broker's environment are concisely addressed; but they are not the focus of this study.

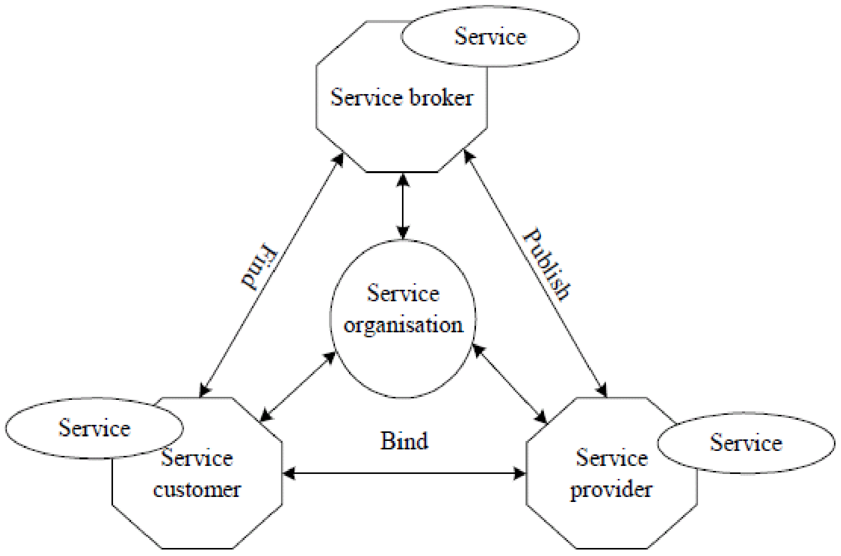


Figure 3.3: SOA Generic Architecture

Traditional Software Engineering (SE) methods are, to a certain extent, not suited for delivering service-oriented systems (Tsai, 2005; van den Heuvel *et al.*, 2009). For example, service-oriented systems have additional activities (such as discovery and composition) and different requirements, when compared to traditional software systems. Furthermore, service-based systems are mainly characterized by SOA design principles, such as loose-coupling, interoperability, composability, discoverability, dynamism, adaptation, and re-usability (Erl, 2008), whereas traditional software systems do not necessarily have to adhere to service design principles.

SOA design and development principles are extensively applied in SOSE, as compared to SE (Anaby-Tavor. *et al.*, 2008). SOSE methods and tools are different from those used in the SE paradigm (Kirda *et al.*, 2001; Sassen & Macmillan, 2005). SOSE focuses on turning business processes into adaptive and composable (Web) services; whereas, SE focuses on the development and maintenance of static and traditional software systems. Figure 3.4 illustrates the relationship between SE and SOSE paradigms. As shown, SOSE extends from SE and Web Engineering (WE).

There are subtle differences and similarities between SOSE, WE and SE. In fact, as illustrated in Figure 3.4, these paradigms are interconnected. For example, SOSE methods inherit and extend some of the methods found in the SE paradigm. In Section 3.2, the main distinctive features between these paradigms are summarised.

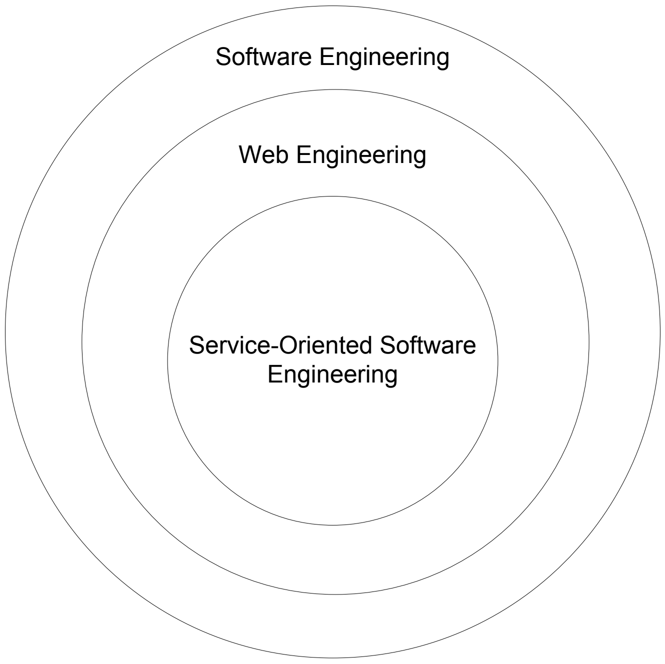


Figure 3.4: Relationship Perspective of Engineering Principles

SOSE can be described as a discipline concerned with a set of activities that deal with *“systematic analysis, design, development, deployment, publication, and execution of service-based systems”* (Cardoso, Voigt & Winkler, 2008). A more precise and comprehensive definition by van den Heuvel *et al.,* (2009) states that SOSE is a *"science and application of concepts, models, methods, and tools to design, develop (source), deploy, test, provide, and maintain business-aligned, and SOA-based software systems in a disciplined, reproducible, and repeatable manner".*

SE can be defined as a *paradigm that deals with all aspects of non-SOA software systems engineering, such as analysis, design, development, testing, implementation, documentation, configuration, and maintenance* (Sommerville, 2006). WE is a systematic process of developing and applying knowledge to engineer quality Web applications (Suh, 2005) and it also extends from SE.

In this chapter, the main focus is on providing a literature overview of the phases involved in service engineering. The SOSE lifecycle approach, as proposed by Zhang, Zhang and Cai (2007) is described. SOSE is core to this study, since the main focus is on the engineering process that could simplify and ease the manner in which intelligent semantic services are designed and developed for publication and consumption by service providers and consumers.

3.2. COMPARISON: SE, WE, AND SOSE

Table 3.1, demonstrates the distinct features of the SE, WE, and SOSE paradigms. These paradigms are compared, using selected key features adopted from Breivold and Larsson (2007), as well as features derived from the objectives of this study.

Table 3.1: Comparison between SE, WE, and SOSE

Features	**SE**	**WE**	**SOSE**
Functional Requirements	Specific	Specific/ Generic	Generic
Project Scope	Large	Varies	Small
Production Time	Long	Varies	Short
Production Costs	High	Medium	Low
Growth and Change	Slow	Fast	Fast
Market	Narrow	Broad	Broad
Platform	Dependent	Independent	Independent
User Interface	Standard	Varies	Varies

In SE, functional requirements are normally specific, as compared with SOSE, where general market requirements or market trends are used as the basis for producing new services (Bicer *et al.*, 2009; Ginige, 2002; Gu & Lago, 2007). Moreover, in traditional software development, software requirements do not change frequently, once the software system has been tested and packaged.

On the contrary, service-based system requirements change rapidly, especially due to the evolution in market trends and requirements (Gu & Lago, 2007). In general, functional requirements in WE can be specific or generic, depending on the intended solution. For example, some Google Web applications (e.g. Google Sites) are designed, based on the generic functional requirements, and enterprise Web applications are mainly engineered, based on specific functional requirements.

Service-based systems have a short production time-span, as compared with legacy software applications; which are engineered over a long period of time, due to their large project scope. On the other hand, the project scope of Web-applications varies, depending on the functional requirements and the complexity of the solution.

Ideally, SOSE is distinguished by small-scale projects (Stojanovic & Dahanayake, 2005:27). Each service is concerned with a specific functionality or capability, such as “currency conversion”; whilst traditional software systems encompass all the major and sometimes redundant functionalities, such as for example: “ordering, invoicing and printing”. Nevertheless, service-based systems could also include multiple capabilities; but generally, additional functionalities would be accomplished by other external composite services possibly developed by different service providers. This is made possible by the re-usability and interoperability aspects of SOA. The assumption is that since the project scope of service-oriented systems tends to be less than that of traditional software systems, the production costs and time of the former would also be lower. This reasoning is the same for Web-based applications; however, the complexity of the solution needs to be considered.

In SOSE and WE, the growth and change of services and the requirements are quite rapid, as compared with systems produced under SE (Ginige, 2002; Stojanovic & Dahanayake, 2005:28). Due to fierce competition, there can be a number of services; and generally Web applications that offer the same capabilities, could actually motivate service providers to always “think ahead” in offering value-added services. However, in SE, changes are usually implemented more slowly than they are needed. This is mainly because SE techniques were not intended to adapt to frequent changes (Stojanovic & Dahanayake, 2005:28).

One of the main goals of SOA is to enable high quality and flexible software production that enables adaptation, re-usability and interoperability (Yu & Ong, 2009). Thus, in SOSE and WE, the target market is usually broad and global, as compared with SE; where users of the system are usually specific, and within a narrow domain.

The other main difference between SE, WE and SOSE is that services and Web applications are platform-independent (Tsai, 2005). This means that service-oriented systems and Web applications could be accessed and executed from any platform that supports services or Web technologies, including mobile devices. In SE, software systems are generally platform-dependent, because systems are usually developed or produced for a specific platform, such as Windows. The software systems developed to run in Windows will thus need to be re-engineered, in order to be implemented under a different platform, such as Linux.

Traditional software applications have standard user interfaces, which, in some cases, are tightly bound to the selected system. In general, services have a common interface description; but these can usually be accessed across various user interfaces and devices – without changing the capability of the service. In WE, user interfaces could be engineered to match different user requirements, as they are, in many cases, not bound to the actual business logic.

Lastly, one other difference between SOSE and SE is that SOSE is solution-driven, whilst SE is product-driven (Stojanovic & Dahanayake, 2005:33). Simply put, SE techniques are generally applied, where complex and sometimes stand-alone computer systems are developed, and the goal of such development is to deliver a complete and functional software product. On the contrary, SOSE is well-suited for modular solutions or independent services; which do not need to form part of a fully functional business software product. Nevertheless, modular services delivered through a SOSE technique could also be composed to realize a complete and functional software product.

3.3. SOSE LIFE CYCLE

Service-Oriented Software Engineering (SOSE), as a relatively new approach, involves different processes and stakeholders. According to Kilian-Kehr (2008), some of the role-players that may be involved in the SOSE process include*:*

- *Service designer: The* provider of service specifications*;*
- *Service producer:* Someone who creates services on behalf of the providers*;*
- *Service provider:* The stakeholder that offers the actual service;
- *Service consumer:* The consumer of available and offered services.

However, in many instances, these stakeholders are grouped into three, namely: service provider (creator), service host (broker), and service consumer (Breivold & Larsson, 2007).

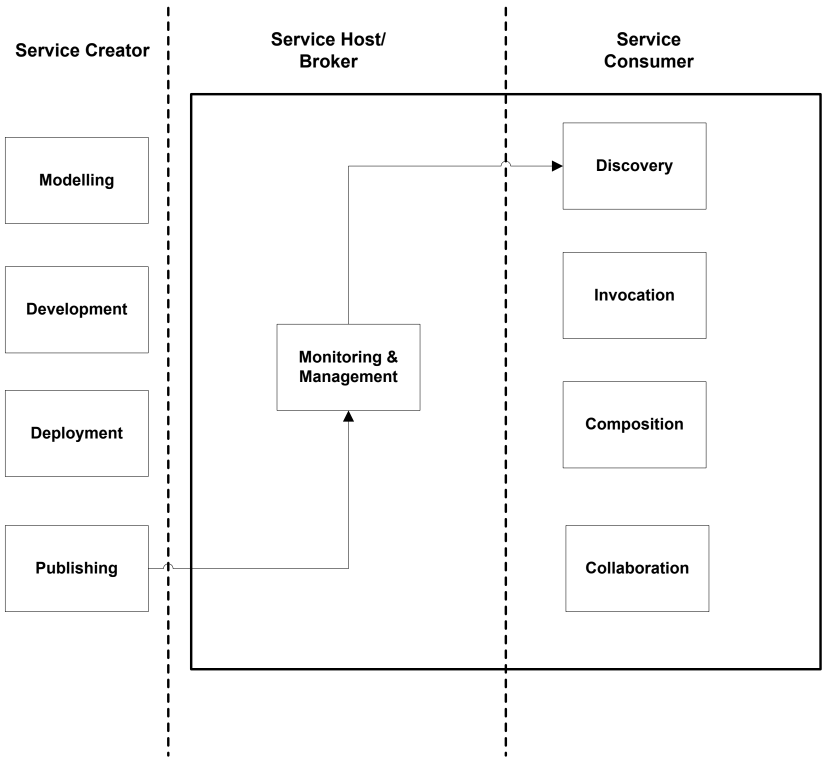


Figure 3.5: SOSE Life Cycle

As depicted in Figure 3.5 (Zhang, Zhang & Cai, 2007), the services lifecycle encompasses several phases, such as service development, service composition, and service management. These phases are also illustrated in Figure 3.6 (Papazoglou & van den Heuvel, 2006). Figure 3.7 illustrates five phases that are generally found in software engineering, namely: *requirements, design, implementation (development), testing or verification, and maintenance* (Yu & Ong, 2009), and these phases are the basis for SOSE.

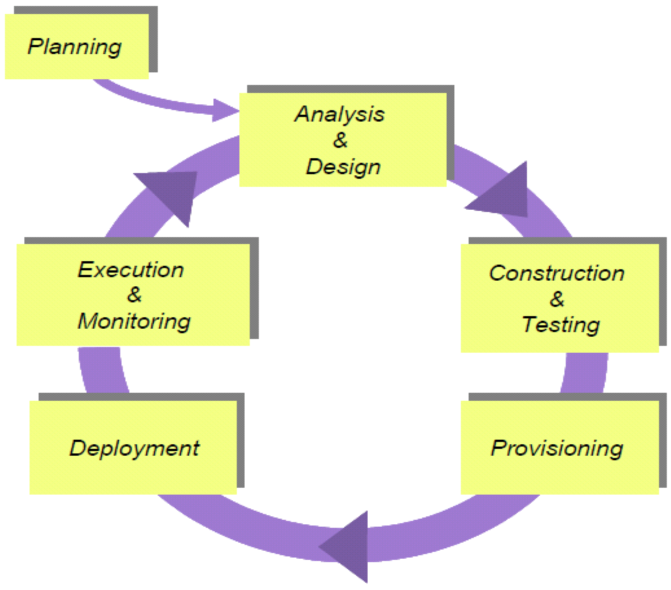


Figure 3.6: Web Service Engineering Life Cycle

In SE, the software development lifecycle (SDLC in Figure 3.7) is common across software development projects. However, in SOSE, researchers have proposed a number of service development lifecycles over the recent past (Gu & Lago, 2007). Although there are some differences between these service development lifecycles, the common foundation is usually the service-oriented principles. Hence, in this chapter, reference is made only to the SOSE lifecycle, as proposed by Zhang, Zhang and Cai (2007) (cf. Figure 3.5).

The SOSE lifecycle is essential for service engineering, as it can aid designers, developers, service brokers, and related stakeholders to have a clear understanding of what activities are involved, when designing, developing, and facilitating the usage of services and service-based systems.

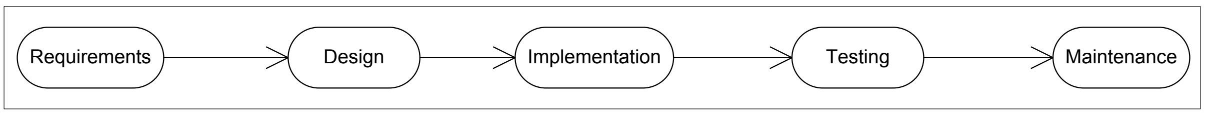


Figure 3.7: Traditional Software Engineering Life Cycle

In the following subsections, a description of each phase of the SOSE lifecycle is provided in detail, with the focus only on the phases of the service creator; as that is where the proposed study is focusing.

3.3.1. Modelling

Services are normally initiated, based on market requirements and trends, rather than on specific client requirements. According to Gu and Lago (2007), SOSE normally starts with a generic market scan, where service providers ascertain the service requirements, by analyzing trends and market demands. Furthermore, an effort is also made to ensure that a service fulfilling a perceived market demand is not already available in various other service marketplaces and repositories. This process can be quite useful in preventing redundant service production. It should always be kept in mind that SOA encourages re-usability (Erl, 2008); where third-party services could be composed with new services to satisfy evolving market requirements.

A model can simply be defined as an abstract representation of a system's behaviour (Stahl & Volter, 2006). According to Gronmo *et al.* (2004), models govern service development, as they can be converted into program code during the development phase; thereby, increasing the speed of service development and deployment. Service models are pivotal to the entire service development process. They capture the problem domain quite clearly, as compared with the actual implementation, which tends to focus heavily on the technological or implementation issues. Additionally, service models are abstract, and mostly concentrate on the main activities or business processes, without focusing on the "how".

In the modelling phase, various techniques to model a service could be applied. For instance, Business Process Modelling Notation (BPMN[14]) and Unified Modelling Language (UML[15]) are some of the common techniques that are used to model specific business processes. UML is more generic; where a number of complex

[14] BPMN is a standardized graphical notation for modelling business processes. It uses various notations for specific events, activities, sequences and relationships between processes. For more information see: http://www.bpmn.org
[15] UML is a standard modelling (specification) language or notation developed by the Object Management Group (OMG) for the purpose of modelling complex systems. For more information see: http://www.uml.org

systems can be modelled at various levels, while BPMN is domain-specific and supports one level of modelling using business process diagrams (BPD).

In concluding this sub-section, it is important to highlight that service modelling can be further divided into three sub-phases, namely: (1) Service identification; (2) service specification; and (3) service realization (Bicer *et al.*, 2009; Yu & Ong, 2009). The identification phase is about determining the goals that are to be accomplished by a service: service specification documents, agreed-upon service operations, and properties. The sub-activity called service realization is concerned with the actual advertisement of the innovated service.

3.3.2. Development

In this phase, a modelled service could be realized using different types of high-level programming languages, such as Java, C#, C++, PHP, and others. In some cases, specific modelling tools could be exploited for generating partial programming codes from the service model. In such cases, developers need only to add the implementation code within the generated code skeletons, thus minimizing service development time.

Once the service capability has been fully implemented, functional service descriptions (i.e. service name, operations, input and output parameters, messaging types) could be manually defined and captured within WSDL documents, or any other service descriptions. This process could also be partially automated by using converters, such as Java2WSDL; this is an Eclipse plug-in that automatically transforms Java classes into WSDL descriptions (Studer, Grimm & Abecker, 2007:312).

Thereafter, programming code stubs of these classes could be generated and supplemented by the developer, based on the service interface defined in WSDL, or in other service description languages. Service developers need to also perform other activities, such as testing, and maintenance to ensure quality control and expected performance levels (Gu & Lago, 2007). These activities are embedded within the generic SE lifecycle as depicted in Figure 3.7.

The phases described in this sub-section are well-suited for producing conventional Web services. When developing syntactic Web services (WS), the focus is on the behavioural issues; and the incorporation of semantic descriptions is not addressed.

For instance, the service models produced in the modelling phase, using UML or BPMN generally do not include any semantic descriptions that could assist in minimizing model interpretation difficulties by different stakeholders. The service descriptions built using WSDL or other conversion tools, such as Java2WSDL are only syntactic. Thus, when it comes to SWS, additional tasks need to be performed by developers during the modelling and developmental phases.

There are developments in this regard, such as using tools (e.g. WSDL2OWL-S) (Studer, Grimm & Abecker, 2007:314) that convert WSDL descriptions to semantic descriptions. However, these tools are not by default integrated into the service development platforms; and they currently have their own challenges (Moulin, Sbodio & Bettahar, 2005), such as the lack of multiple-language support, uniformity, and completeness when for example translating syntactic descriptions to semantic descriptions.

3.3.3. Deployment

The deployment phase is about activating the constructed services for consumption. The deployment process can be compared to a process of uploading a service into the Web server. The deployment stage in the SOSE life cycle gives the developer an opportunity to actually test the performance of the developed service (Gu & Lago, 2007).

This process might also be recursive, as the developer would need to be satisfied that the service performs as intended, before advertising the service for public usage. As stated by Zhang, Zhang, and Cai (2007:104), the deployment phase also involves the binding of functional service descriptions to service protocols, such as SOAP over HTTP. Once a service has been deployed, it can only be invoked and consumed by the provider. This means that the public community would not be able to access the service until it is advertised or published to appropriate public marketplaces or service registries.

3.3.4. Publishing

This phase mainly involves advertising the service for public access, invocation, and execution. Information, such as how to invoke and execute the developed service, is published in a public service registry, through the use of service descriptions. Figure 3.8 demonstrates how services – through service descriptions – are published and discovered using messaging protocols, such as SOAP (Newcomer, 2004:31). Service registries are managed and monitored by service brokers (Zhang, Zhang & Cai, 2007:104).

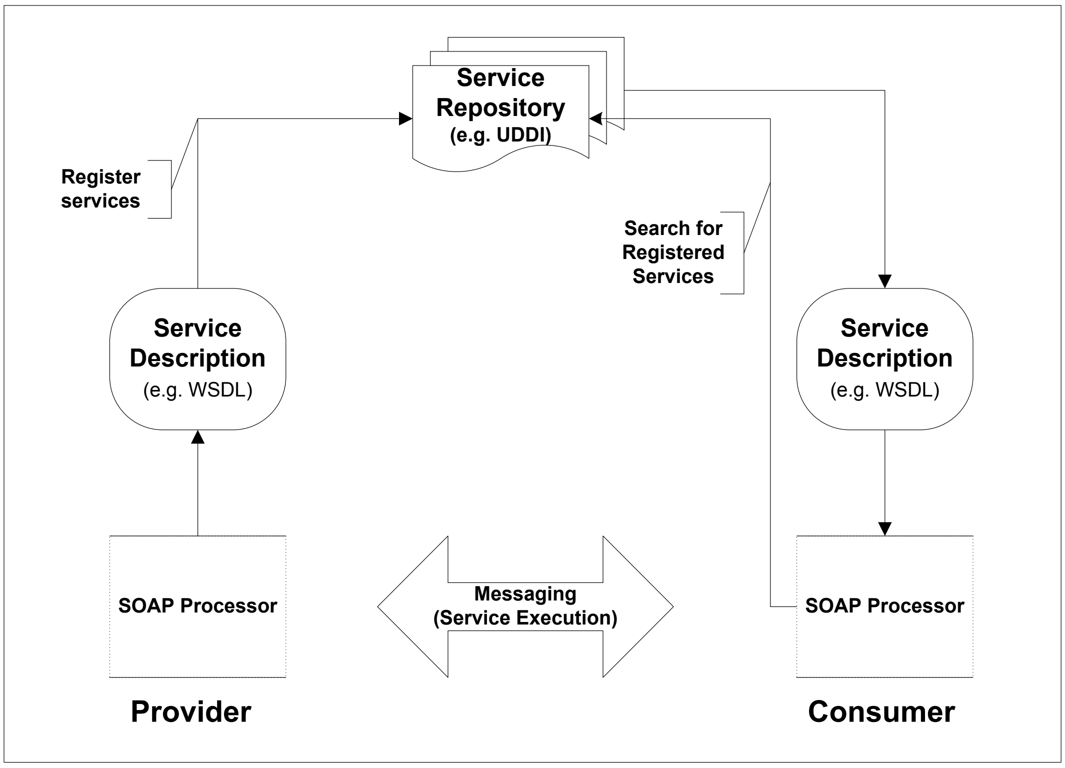

Service Repository (e.g. UDDI)
Register services
Search for Registered Services
Service Description (e.g. WSDL)
Service Description (e.g. WSDL)
SOAP Processor
Messaging (Service Execution)
SOAP Processor
Provider
Consumer

Figure 3.8: Service Publication and Discovery

In concluding Section 3.3, it is essential to be aware of, and to properly manage, the different phases in the SOSE life cycle. Furthermore, a variety of techniques and tools are available to achieve most of the activities described above. Approaches, such as *model-driven* engineering (MDE) (Anaby-Tavor. *et al.*, 2008), and *code-*

driven engineering (CDE) (Srinivasan, Paolucci & Sycara, 2006) could be used as alternatives.

MDE focuses on models, meta-models, and transformations, as cornerstones for SOSE. In this approach, the engineering process commences with the creation of models at different levels of abstraction (Anaby-Tavor. *et al.*, 2008), in order to carefully design, analyze, and capture the requirements, behaviours, and structure of the intended service or service-based system. MDE is commonly preferred in service development as models enable developers and researchers to deal with various concerns before the actual system is built and implemented; thereby, reducing the risks of system failure and collapse during execution.

A code-driven engineering approach tends to be favoured by developers who are mainly interested in the implementation or prototyping of systems. According to Srinivasan, Paolucci, and Sycara (2005), the code-driven approach starts by implementing a service – using a particular programming language. Service descriptions and models are then be derived from the implemented code.

Lastly, for developers to realize individual services, and service-based systems, a number of phases need to be completed. However, within the SOSE domain, there is a lack of simple and unified platforms and tools that support the process of engineering semantic services, as compared with the engineering of conventional Web services. Equally so, it is a challenge for service designers and providers to manually complete each SOSE lifecycle phase error-free, without simple, efficient, and interoperable software tools. One of the key objectives of this thesis is to address some of these challenges.

3.4. SUMMARY

The overall focus of this study is primarily on the modelling, and development (this includes service creation, semantic descriptions and annotations, as well as intelligence wrapping) of intelligent semantic services. We have realized that traditional software engineering techniques cannot be directly applied when building semantic services. Hence, the concept of Service-Oriented Software Engineering (SOSE) has been introduced in this chapter.

Semantic services and service-oriented systems have different life cycles compared to traditional software systems. For example, in service engineering, a service can be searched, discovered, selected, composed, invoked, executed, and so forth, whilst the lifecycle of a traditional software system is different, with fewer phases.

A comparison of SOSE, WE and SE has been provided, to establish the main differences between these approaches. This was accomplished by taking into consideration that service engineering inherits and extends software and Web engineering techniques. The SOSE lifecycle was presented, in order to clarify the main phases on which the proposed study focuses - when considering phases within the service engineering process.

Furthermore, the SOSE life cycle also solidifies our research argument that without supporting methods and tools, it could be a challenge to produce efficient services, when manually handling the SOSE activities. This was done by detailing the different activities conducted when engineering service-oriented systems. From the different phases, it is appropriate to suggest that manual processes are not enough to simplify and accelerate the process of engineering semantic services. There is a need for novel methods and tools that could promote and unify service modelling, development, deployment, re-usability, interoperability, and even more so, tools that could deal with the complexities involved when engineering service-oriented systems.

In Chapter 4, we shall discuss the prominent models, methods, and tools used in semantic SOSE for realizing and supporting ontology definition, semantic descriptions, and semantic annotations.

CHAPTER 4: Semantic Service Models and Related Tools

The literature review is concluded in this chapter. Prominent semantic models and semantic description languages relevant to the engineering of semantic services are reviewed and discussed. This includes common related tools that have come forth over the recent years in an attempt to ensure that semantic services become a reality.

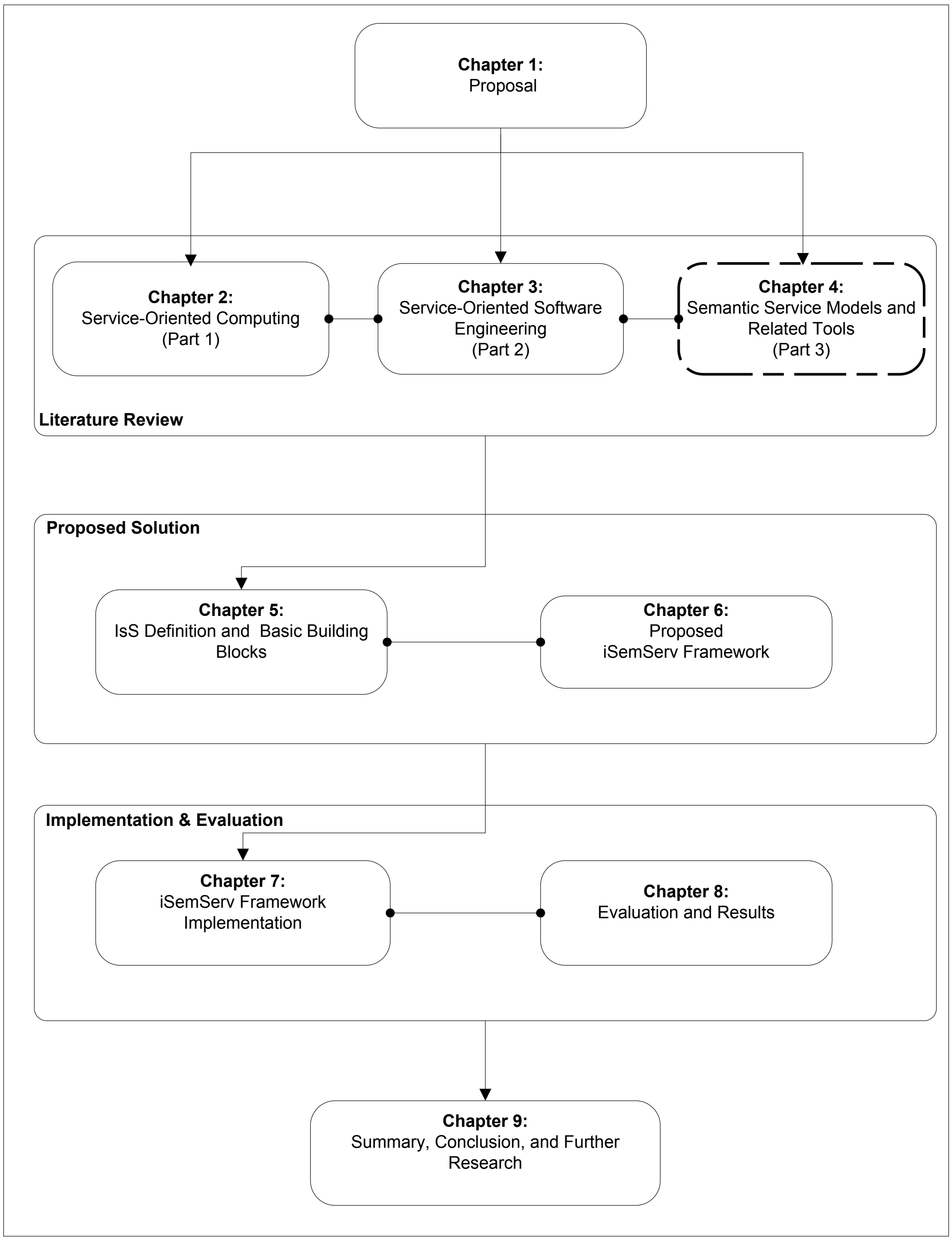

Chapter 1:
Proposal
Chapter 2:
Service-Oriented Computing
(Part 1)
Chapter 3:
Service-Oriented Software Engineering
(Part 2)
Chapter 4:
Semantic Service Models and Related Tools
(Part 3)
Literature Review
Proposed Solution
Chapter 5:
IsS Definition and Basic Building Blocks
Chapter 6:
Proposed iSemServ Framework
Implementation & Evaluation
Chapter 7:
iSemServ Framework Implementation
Chapter 8:
Evaluation and Results
Chapter 9:
Summary, Conclusion, and Further Research

Figure 4.1: Overall Thesis Structure

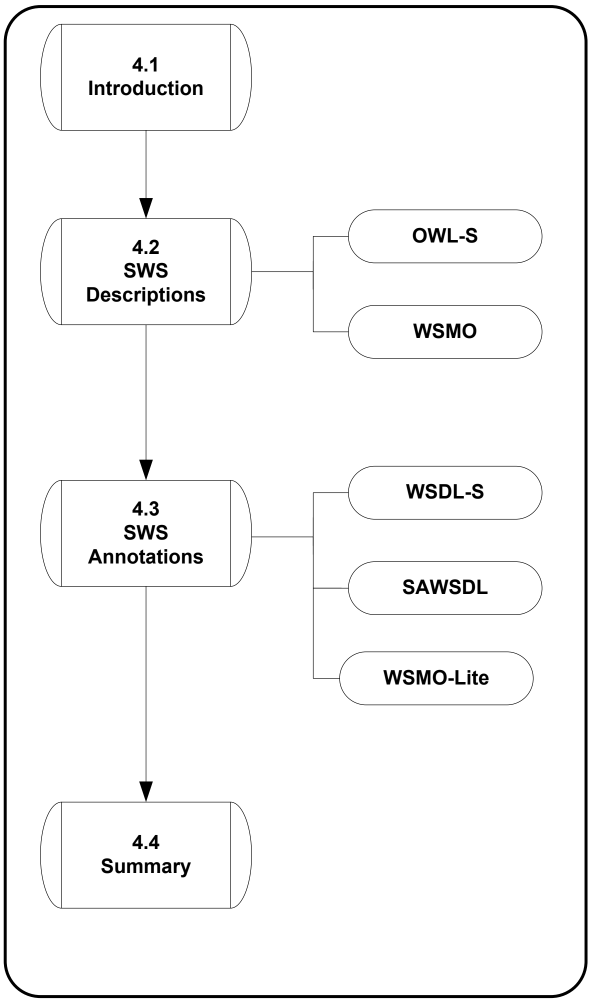


Figure 4.2: Chapter 4 Layout

4.1. INTRODUCTION

The SOSE lifecycle described in Chapter 3 is suitable for developing Web services (WS). However, it is not fully suitable for engineering Semantic Web Services (SWS). This is due to the fact that the development of SWS requires additional steps, such as ontology and semantic descriptions development. Nevertheless, the traditional SOSE lifecycle phases (Zhang, Zhang & Cai, 2007) could be applied for engineering some components of SWS. However, this needs to be supported by developing additional novel methods, standards, and platforms that could facilitate other activities involved in engineering SWS, such as the development of domain ontologies and semantic descriptions.

The field of WS has evolved over the years, and the process of engineering WS has greatly improved over time. This could mainly be attributed to mature methods, tools, and platforms. Hence, the argument that is put forth in this study is that for SWS to reach similar levels of success as WS, supporting methods, tools, and unified platforms are of importance.

In the following sections, current advances and challenges in the field of SWS are discussed. This includes the overarching semantic service models primarily focused on semantic descriptions and domain ontologies, including available standards, languages, tools, and platforms that attempt to deal with the issues of simplifying the process of engineering semantic services.

4.2. SWS DESCRIPTIONS

In Chapter 2, SWS and related concepts were briefly introduced. In this section, we elaborate on the essential building blocks of semantic services - semantic descriptions, and related ontology models. Ontologies and semantic descriptions are core to the process of developing semantic services because they enable services to be machine-interpretable and machine-processable.

As noted in Section 2.2.1 of Chapter 2, WSDL as a standardized XML-based language that is used for syntactically describing Web services. WSDL describes a number of service aspects, such as service name, data types, operations, input and output parameters as well as message types – all intended for service advertisement, discovery, and invocation (Yu, 2007:208). However, WSDL only provides syntactic descriptions; and these descriptions, as mentioned in Chapter 1 and Chapter 2; do not enable WS to be intelligently processed with minimal user intervention. The concept of semantic descriptions is thus intended to address these challenges.

Accordingly, there are a number of ontology-based models and languages that have emerged over the recent past for facilitating the construction of semantic descriptions, such as Web Ontology Language for Service (OWL-S), Web Services Modelling Ontology (WSMO) (Kashyap, Bussler & Moran, 2008:259) and WSMO-Lite (Vitvar, Kopecky & Fensel, 2009).

In the following subsections, we report on the prevalent heavy-weight semantic description models and their underlying languages. These are: OWL-S and WSMO (Acuna & Marcos, 2006; Lia, Abela & Scicluna, 2009; Wang *et al.*, 2007). In Section 4.3, WSDL-S, SAWSDL, and WSMO-Lite are highlighted as some of the existing lightweight approaches for annotating SWS.

4.2.1. OWL-S

As a semantic description language, Web Ontology Language for Services (OWL-S) is based on OWL (Web Ontology Language) and RDF (McGuinness & van Harmelen, 2004); and these comprise the common basis for the Semantic Web (Kashyap, Bussler & Moran, 2008:259) and SWS.

The main objective of OWL-S is on the enablement of services that could be automatically discovered, composed, invoked, and executed by software agents and users respectively (Martin *et al.*, 2004). OWL-S, formerly known as DARPA Agent Mark-up Language for Services (DAML-S), was initiated by DAML[16], and is supported by W3C[17]. It is structured into three elements, namely: the *Service Profile,*

[16] DAML is a DARPA Agent Mark-up Language programme with an objective to develop a language and tools to facilitate the concept of the Semantic Web. For more information see http://www.daml.org
[17] W3C is a World Wide Web Consortium responsible for the standardization of Web technologies. For more details see http://www.w3.org

the Service Model or Process Model, and Service Grounding, as depicted in Figure 4.3 (Martin *et al.*, 2004).

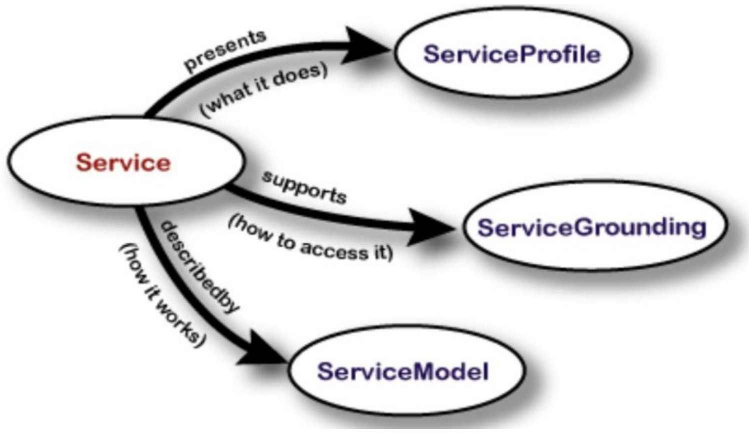


Figure 4.3: OWL-S top level service ontology

The *Service Profile* advertises the information necessary for semantic service discovery. The information presents "what a service does". This is attained through a profile class that defines the capabilities of the service by specifying both functional (i.e. inputs, outputs, preconditions, and effects) and non-functional properties (i.e. service name, textual service description, contact information, and service category) (Elenius *et al.*, 2005; Martin *et al.*, 2004). According to Martin *et a*l. (2004), a host of non-functional properties can be used to describe a variety of features of a particular service, such as service rating, estimated response time, and geographic scope.

In OWL-S, each service is represented through the instantiation of the *Service* concept, as shown in Figure 4.3. The instantiated *Service* acts as a point of reference for semantically describing a Web service (Lara *et al.*, 2004; Martin *et al.*, 2004)- by using the three elements of OWL-S. Listing 4.1shows an excerpt of a Service Profile class – describing inputs, outputs, and non-functional properties of a service named "BravoAir_ReservationAgent".

```
1:<rdf:RDF xml:base="http://www.daml.org/services/owl-s/1.1/BravoAirProfile.owl">
2:<profileHierarchy:AirlineTicketing rdf:ID="Profile_BravoAir_ReservationAgent">
3:<!-- reference to the service specification -->
4:<service:presentedBy rdf:resource="http://www.daml.org/services/owl-s/1.1
BravoAirService.owl#BravoAir_ReservationAgent"/>
5:<!-- reference to the process model specification -->
6:<profile:has_process rdf:resource="http://www.daml.org/services/owl-s/1.1/BravoAirProcess.owl#BravoAir_Process"/>
7:<profile:serviceName>BravoAir_ReservationAgent</profile:serviceName>
8:<profile:textDescription>
9:This  service provide flight reservations based on the specification of a flight request.  This typically involves a
10:departure airport, an arrival airport, a departure date, and if a return trip is required, a return date. If the desired flight
11:is available, an itinerary and reservation number will be returned.
12:</profile:textDescription>
13:<profile:serviceParameter>
14:<addParam:GeographicRadius rdf:ID="BravoAir-geographicRadius">
15:<profile:serviceParameterName>
16:          BravoAir Geographic Radius
17:        </profile:serviceParameterName>
18:<profile:sParameter rdf:resource="http://www.daml.org/services/owl-s/1.1/Country.owl#UnitedStates"/>
19:</addParam:GeographicRadius>
20:</profile:serviceParameter>
21:<profile:hasInput rdf:resource="http://www.daml.org/services/owl-s/1.1BravoAirProcess.owl#DepartureAirport"/>
22:<profile:hasInput rdf:resource="http://www.daml.org/services/owl-s/1.1/BravoAirProcess.owl#ArrivalAirport"/>
23:<profile:hasOutput rdf:resource="http://www.daml.org/services/owl-s/1.1/BravoAirProcess.owl#FlightsFound"/>
24:</profileHierarchy:AirlineTicketing>
25</rdf:RDF>
```

Listing 4.1: Excerpt of the OWL-S Service Profile

As shown in Listing 4.1, the Service Profile is structured using specialized XML tags, such as `<profile:textDescription>`[18] used to encapsulate non-technical information that describe the offerings of a semantic service, and `<profile:serviceParameter>`[19] that captures a list of properties that supplement the profile class for service discoverability (Martin *et al.*, 2004).

The *Service Model* provides a detailed description of service operations, such as how a service could be executed by the requester, and how it performs its activities, including data flow and message control between Web methods (Balzer, Liebig & Wagner, 2004). Furthermore, the Service Model derives the functional properties that are used in the profile class (Elenius *et al.*, 2005). Additionally, it is formed, on the basis of one or more process models defined by OWL-S for executing discovered services. These are: *atomic processes*, *simple processes*, and *composite processes* (Kashyap, Bussler & Moran, 2008:260).

[18]See Line 8-12
[19]See Line 13-20

An atomic process can be used by the service discoverer to directly invoke a service containing only a single Web method (Balzer, Liebig & Wagner, 2004). Listing 4.2 depicts an example of a Service Model containing one atomic process that is responsible for getting flight details, such as the Departure Airport, and Outbound Date.

```
<process:AtomicProcess rdf:ID="GetDesiredFlightDetails">
<process:hasInput rdf:resource="#DepartureAirport"/>
<process:hasInput rdf:resource="#ArrivalAirport"/>
<process:hasInput rdf:resource="#OutboundDate"/>
<process:hasInput rdf:resource="#InboundDate"/>
<process:hasInput rdf:resource="#RoundTrip"/>
<process:hasOutput rdf:resource="#FlightsFound"/>
</process:AtomicProcess>
```

Listing 4.2: OWL-S Service Model Class

Simple processes are not meant to be directly invoked; and they are only intended for specifying "abstract views of concrete processes by hiding certain inputs, outputs, preconditions and effects" (Balzer, Liebig & Wagner, 2004). Composite processes are the ones that have multiple steps. They provide for the maintenance of states and messages that can be passed to other Web methods of separate services. Composite processes deal with more than one process; these can be atomic, simple, or even composite (Martin *et al.*, 2004). However, it should be noted that composite processes are composed of atomic processes.

Service Grounding provides semantic descriptions on how clients should communicate or exchange messages with discovered services. Basically, grounding defines how a service is invoked and executed. It binds parameters (i.e. inputs and outputs) defined in the Service Model with concrete parameters and messages defined in syntactic descriptions (Balzer, Liebig & Wagner, 2004; Elenius *et al.*, 2005; Kashyap, Bussler & Moran, 2008:259). An atomic process in the Service Model is linked with operations or Web methods defined in the WSDL document or any other service description document. Inputs and Outputs specified in the process model are linked with service input and output parameters, as defined in the syntactic description document.

Other OWL-S details, such as data types for inputs and outputs and message protocols, are also linked with the same information, as described in syntactic descriptions (Balzer, Liebig & Wagner, 2004). Nevertheless, *Service Grounding* is also capable of describing additional information in OWL-S, such as supported transport protocols, supported message formats, and other low-level information, such as WSDL operations (Elenius *et al.*, 2005; Yu, 2007:249).

OWL-S, as one of the first initiatives for semantically describing services, is interoperable, in the sense that its different elements (e.g. Service Profile) could be re-used by other services. In addition, OWL-S elements are extensible, especially through sub-classing (Lara *et al.*, 2004). For instance, a specific Service Model could be extended to address additional behavioural situations in semantically described services (Elenius *et al.*, 2005).

Nevertheless, OWL-S (i.e. OWL-S 1.1) has its own shortcomings. According to Wang *et al.* (2007), the reasoning capabilities of OWL-S are weak, particularly due to the lack of matured reasoners within the OWL-S domain. Moreover, OWL-S is not capable of adequately expressing a complete list of service's non-functional properties, such as availability and performance. In addition, OWL-S also does not explicitly provide approaches for handling heterogeneity issues (de Bruijn *et al.*, 2005b) between the different elements (i.e. Service Profile, Service Model, and Service Grounding).

OWL-S is complex and resource intensive from the perspective of an average service developer. More than this, OWL-S has a steep learning curve for service development experts. As a result, it falls short in facilitating the realization of SWS in a simpler and quicker manner. The complexity challenges that come with OWL-S are further compounded by the lack of adequate tools that could ease the development of such semantic services (Agre *et al.*, 2007; Balzer, Liebig & Wagner, 2004). Nevertheless, there are tools that exist to partially support developers with the creation of ontologies, such as OWL-S IDE (Srinivasan, Paolucci & Sycara, 2006) and Protégé (Horridge *et al.*, 2007).

However, some of these tools are not integrated with existing service engineering platforms. Even those that could be easily integrated into existing development platforms, such as Eclipse; do not support the engineering of semantic services that are intelligent beyond the use of ontologies. Hence, in this study, our aim is to demonstrate a proof of concept platform that will not only simplify and accelerate the process of engineering intelligent semantic services, but will also provide an environment that is unified, and useful to both experts and non-experts.

The following sub-section briefly elaborates on some of the current tools that can be used to create semantic descriptions using OWL-S.

4.2.1.1. OWL-S Tools

One of the prominent tools that facilitates the development, advertisement, and consumption of OWL-S based services, and supports a complete life cycle of SWS, is OWL-S IDE (formerly known as CODE) by Srinivasan, Paolucci, and Sycara (2005). OWL-S IDE integrates the semantic description process and the service capability implementation process within one environment. OWL-S IDE extends currently existing WS tools and standards, such as UDDI, in order to ensure seamless development, advertisement, and the consumption of semantically based services. OWL-S IDE is based on the Eclipse plug-in environment.

OWL-S IDE, as the name suggests, is coupled to OWL; it does not accommodate other ontology models, such as WSMO; and graphical representation of ontologies or services is not supported (Elenius *et al.*, 2005).

One of the other tools that promotes OWL-S descriptions development is called OWL-S Editor[20] (Elenius *et al.*, 2005); this is also used by the OWL-S IDE as described above. OWL-S Editor only focuses on ontology editing, and does not integrate any service programming environment within its platform. It is incorporated

[20] OWL-S Editor is an open source tool and is available under http://owlseditor.semwebcentral.org

in the Protégé platform on top of the OWL Ontology Editor plug-in. It facilitates the development of various domain and service ontologies. However, it is not suited for handling multiple and heterogeneous domain and service ontologies.

Other individual tools for facilitating the development of OWL-S services include converters, such as WSDL2OWL-S[21] and Java2OWL-S (Studer, Grimm & Abecker, 2007:312). WSDL2OWL-S converts WSDL descriptions into partial OWL-S classes. These are: *Service Profile, Service Model*, and *Service Grounding* classes. WSDL2OWL-S usually comes as part of the OWL-S toolset; and it could also be incorporated into environments, such as Eclipse.

Java2OWL-S[22] is responsible for partially translating Java classes into OWL-S profile, process model, and service grounding classes. As with WSDL2OWL-S, the service grounding is completely generated, whilst the service profile and service model are partially generated. Java2OWL-S combines Java2WSDL[23] and WSDL2OWL-S converters to support the translation of Java classes to OWL-S classes (Studer, Grimm & Abecker, 2007:314).

4.2.2.WSMO

WSMO is a conceptual ontology model. It was developed by DERI (Digital Enterprise Research Institute)[24] (Acuna & Marcos, 2006; de Bruijn *et al.*, 2005a). It is based on the Web Service Modelling Language (WSML) (Cabral *et al.*, 2004; Fensel & Bussler, 2002). Various elements that can be semantically described using WSMO are *Ontologies, Web services, Goals, and Mediators.*

[21] WSDL2OWL-S converts WSDL documents to OWL-S ontology specifications, and can be downloaded at http://www.daml.ri.cmu.edu/wsdl2owls/ or http://projects.semwebcentral.org/projects/wsdl2owl-s/
[22] Java2OWL-S can be downloaded from http://projects.semwebcentral.org/projects/java2owl-s/, and it uses WSDL2OWL-S in the background and another component called Java2WSDL.
[23] Java2WSDL takes a Java class as input and generates a WSDL description file that can be used to invoked methods as Web services by service requesters. It is part of Apache Axis. More information can be found on http://ws.apache.org/axis/
[24] See http://www.deri.org/

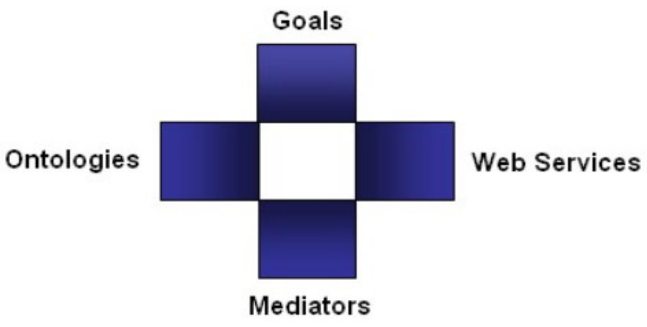
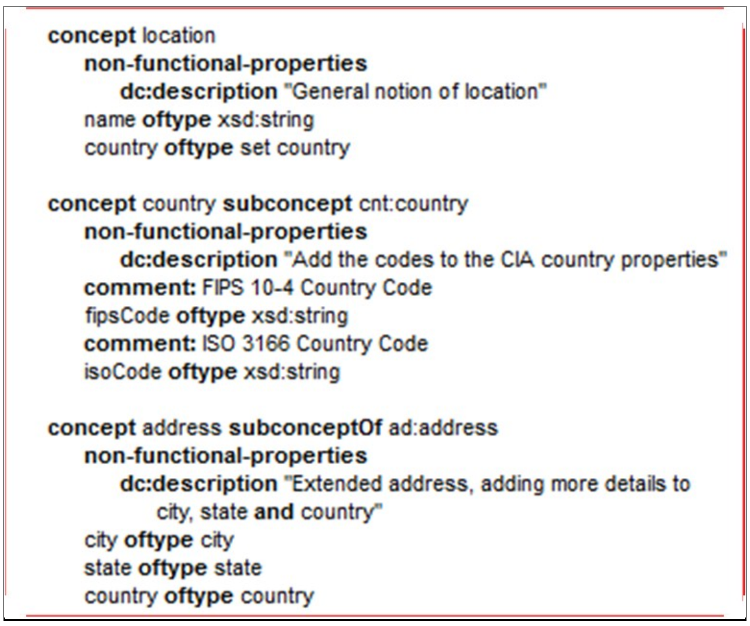


Figure 4.4: Top-level WSMO Elements

As depicted in Figure 4.4, the core elements of WSMO are described according to the WSMO submission document (de Bruijn *et al.*, 2005a) to the W3C as follows:

- **Ontologies:** provide common terminologies and knowledge representations (i.e. domain and service ontologies) that could be used to achieve an understanding between Web services and Goals, including other core elements of WSMO for interoperability purposes. Listing 4.3 shows a partial domain ontology describing locations, such as continents, countries and cities and their interrelations using the free-format of WSML.



```
concept location
    non-functional-properties
        dc:description "General notion of location"
    name oftype xsd:string
    country oftype set country

concept country subconcept cnt:country
    non-functional-properties
        dc:description "Add the codes to the CIA country properties"
    comment: FIPS 10-4 Country Code
    fipsCode oftype xsd:string
    comment: ISO 3166 Country Code
    isoCode oftype xsd:string

concept address subconceptOf ad:address
    non-functional-properties
        dc:description "Extended address, adding more details to
            city, state and country"
    city oftype city
    state oftype state
    country oftype country
```

Listing 4.3[25]**:** WSMO Ontology example

[25] The complete ontology can be found at: http://www.wsmo.org/ontologies/location/

As it can be noted, the information captured by the example (Listing 4.3) can vary depending on the domain and the number of concepts and relations involved within that particular domain.

- **Goals:** represent objectives or intentions in WSML that the service requester expects to be accomplished by the Web service. Goals are usually represented in terms of functional and non-functional requirements (Roman et al., 2006).
- **Web services[26]:** provide ontological descriptions that define functional, non-functional, and behavioural aspects of the service itself. The descriptions can be used by software programs to automatically discover services that are of interest.
- **Mediators:** capture the domain knowledge that is useful for handling interoperability and incompatibility issues between the core WSMO elements. There are four types of mediators supported by WSMO. Firstly, *ooMediators* deal with interoperability issues between different ontologies. Secondly, *ggMediators* connect different Goals and also handle interoperability issues between Goals. Thirdly, *wgMediators* handle cooperation issues between Web services and Goals. Lastly, *wwMediators* handle the interaction and interoperability challenges between co-operating Web services.

As noted above, WSMO uses WSML, a formal language that allows for syntactic and formal specification of different aspects of WSMO. The formal syntax and semantic descriptions provided by WSML can be used to describe different WSMO core elements (de Bruijn *et al.*, 2005c).

Overall, the main differences between OWL-S and WSMO are (Cardoso, 2007a):

- OWL-S is based on OWL, and WSMO is based on WSML.
- OWL-S does not consider challenges of heterogeneity. WSMO provides mediators for dealing with interoperability and heterogeneity problems.
- OWL relies only on Description Logics[27] (DL) (Horrocks & Sattler, 2002), whilst WSML is based on different logical formalisms, such as DL, First Order

[26]Please note in this context, the term Web services refers to ontological descriptions in WSMO, and not distributed Web services

Logic (FOL)[28], and Logic Programming[29] (LP). These facilitate the enablement of formal meaning in semantic descriptions (de Bruijn *et al.*, 2005c; Roman *et al.*, 2006).

The key differences between OWL-S and WSMO are further summarized in Table 4.1, using logical formalisms, ontology language, syntax, and support of heterogeneous domain ontologies as elements of comparison.

Table 4.1: Differences between OWL-S and WSMO

Criteria	OWL-S	WSMO
Logical formalisms	DL	DL, FOL, LP
Ontology language	OWL, RDF	WSML
Syntax	XML	Human-readable syntax XML
Heterogeneity	Not supported	Mediators

WSML has various benefits that are seen as improvements over OWL. All WSML variants use normative human-readable syntax (de Bruijn *et al.*, 2005c). In addition, WSML separates conceptual modelling and logical modelling. Moreover, in WSMO, Semantic descriptions are generated based on well-established logical formalisms, such as Description Logics and First Order Logic. Heterogeneity issues in WSMO are handled through the use of different types of mediators that are specific to WSMO core elements (Roman *et al.*, 2006).

The main drawbacks of WSML and WSMO pertain to the development tools. Tool support, for creating semantic descriptions using WSML is lacking. Additionally, WSML has a steep learning curve, especially when having no background knowledge on logical formalisms when defining logical expressions (e.g. axioms).

[27] Description Logics (DL) are a variety of formal knowledge representation languages intended for modelling concepts, roles and individuals, as well as their relationship. They are extensively used in Artificial Intelligence and have been adopted in the implementation of Semantic Web and Semantic Web Services for knowledge representation, expression, and reasoning.
[28] First Order Logic (FOL), as a logical formalism, provides syntax for formally expressing objects, relations, and functions.
[29] Logic programming (LP) is a family of high-level knowledge representation languages that are commonly used in Artificial Intelligence (AI) for expressing logical properties with regard to computations. Prolog is one of the established Logic Programming languages.

Nevertheless, there are some tools and platforms that have been in the public domain, such as WSMT (Web Service Modelling Toolkit) (Kerrigan *et al.*, 2007), and WSMO Studio (Dimitrov *et al.*, 2007), for supporting the process of knowledge representation, but not the complete end-to-end process of building intelligent semantic services.

However, most of these tools are complex; and they cater for expert developers in the field of SWS. Thus, it can be a challenge for average developers and service providers to easily use these tools to simply and rapidly engineer their business services, as intelligent semantic services. Similar to OWL-S, WSMO description creation tools do not support the development of intelligent semantic services.

4.2.2.1. WSMO Tools

In this section, we discuss some of the tools and platforms that are available to facilitate the process of building WSML compliant ontologies and semantic descriptions. WSMT and WSMO Studio are two of such prominent platforms available in the public domain. However, these tools do not come readily integrated within existing service platforms. In our view, the approach of delivering semantic tools in segregation could also be a barrier for adoption, and for usage by early adopters. Developers always prefer to perform tasks that are related within one integrated space (Rivières & Wiegand, 2004). Some of the WSMO tools are discussed below:

- **WSMT**: Web Service Modelling Toolkit[30] (Kerrigan *et al.*, 2007) provides a graphical environment for creating domain knowledge, using WSML to describe and represent all the core elements of WSMO. It provides support for building WSMO descriptions (i.e. Ontologies, Web services, Goals, and Mediators) and mappings between different mediators. It also interfaces created descriptions with the Semantic Execution Environments (SEEs), such as WSMX (Web Service Execution Environment) (Kerrigan *et al.*, 2007; Roman *et al.*, 2006). The reasoning of the created ontologies is supported,

[30] WSMT is an open source tool and is released under multiple free software licences, such as General Public Licence (GPL) and Lesser General Public Licence (LGPL)

and is handled by the WSML2Reasoner plug-in, which is integrated within the WSMT environment.

WSMT proponents contend that this modelling toolkit minimizes the challenges of creating SWS applications by providing a unified toolset for SWS (Kerrigan *et al.*, 2007). WSMT could be also integrated to the Eclipse environment and extended to include additional toolsets that could aid developers in seamlessly building SWS applications. However, WSMT is mainly developed for SWS experts; and it has a steep learning curve for average developers. This is mainly due to the fact that creating semantic descriptions using low-level lexical notations, as supported by conceptual models, such as WSMO, remains nevertheless a daunting task (Torres, Pelechano & Pastor, 2006).

Contrary to the objectives of this study, WSMT does not systematically, and by default support the actual engineering of semantic services; but it facilitates only the process of building semantic descriptions.

- **WSMO Studio**: is an open-source ontology editor that can be used to specify, using WSML, all the core elements of WSMO (Dimitrov *et al.*, 2007). It is based on the Eclipse framework, and could also be extended with additional plug-ins for re-usability and extensibility purposes (Feier *et al.*, 2005). It supports the annotation of services from WSML descriptions through the SAWSDL implementation, which is discussed in Section 4.3.2.

  It should be noted that although WSMT supports graphical representation and the visualization of ontologies, WSMO studio is core in assisting the developer with building and editing ontologies at the highest level. Nevertheless, visualization tools for different WSML species could always be integrated where needed (Kashyap, Bussler & Moran, 2008:142). In addition to the ontology editing role, WSMO studio supports additional important activities: validation and reasoning of created WSML ontologies, export and import from different WSML variants, support definition of WSMO choreography interfaces through a choreography editor, and facilitates semantic annotation, through the SAWSDL editor (Dimitrov *et al.*, 2007).

Furthermore, WSMO studio supports repositories for storing and querying WSML compliant ontologies, service discovery, and reasoning over ontologies using the WSML2Reasoner[31] (Dimitrov *et al.*, 2007).

- **WSDL2WSMO:** translates WSDL specifications into the corresponding, but partial, WSMO ontology specifications. This tool only generates one core element of WSMO, that is, Ontologies (Bouhissi, Malki & Bouchiha, 2006). The developer would have to use other tools, such as WSMT, to create ontological specifications for other elements, such as Web services, Goals, and Mediators.

- **WSMO4J:** is made up of a group of Java libraries that could be used to parse semantic descriptions created, using WSML to Java class objects (Kashyap, Bussler & Moran, 2008:275). This approach is a top-down approach, where ontologies are defined before actual service implementation. The top-down approach is considered efficient in software development, as it focuses on completeness and understanding with regard to semantic descriptions. Nevertheless, WSMO4J has a steep learning curve, due to its complexity; and the libraries do not provide any form of simplification or complexity hiding.

- **WSMX:** The Web Service Execution Environment is a SOA-based middleware for handling and supporting various aspects of SWS, such as automatic discovery, selection, mediation, composition, invocation, and execution (Roman *et al.*, 2006). It is an open source-based reference for the implementation of WSMO, and it mainly deals with WSMO-based services.

  WSMX is characterized by component decoupling; where components, such as the discovery engine and data mediator are separated, according to their specific functional responsibilities (Facca, Komazec & Toma, 2009).

[31] WSML2Reasoner is available under http://devi.deri.at/wsml2reasoner/

WSMX is seen as a promising and flexible WSMO implementation. It also supports the interoperability and extensibility requirements with the use of plug-ins, where even the generic WSMX components could be exchanged with similar components provided by third parties (Herold, 2008; Roman *et al.*, 2006). However, in the proposed study, WSMX is viewed as relevant after the actual implementation and deployment of semantic services. It could be useful in supporting activities, such as service discovery, service composition, and service execution, which are beyond the scope of this thesis.

4.3. SWS ANNOTATIONS

There have been a number of interventions in the practical realization of semantic services. In the previous section, two high-level and common initiatives (i.e. OWL-S and WSMO) for facilitating the development of semantic descriptions are discussed. However, these solutions are complex and resource intensive. In the following subsections, a brief review of alternative light-weight semantic annotation approaches is provided. The description is limited to WSDL-S, SAWSDL, and WSMO-Lite only, due to their widespread adoption and popularity in the SWS research domain.

4.3.1.WSDL-S

WSDL-S is a lightweight annotation standard not dependent on any specific ontology language or semantic description language (Yu, 2007:266). It extends WSDL descriptions by annotating them with ontological concepts, such as inputs, output, preconditions, effects, and operations (Stollberg, Hepp & Fensel, 2010). Since WSDL-S extends WSDL descriptions with semantic annotations, it could be executed in the WSDL environment (Hernandez, 2007); thus, there is no need for an execution environment, such as WSMX in a case of WSMO-based services. The main objective of WSDL-S is to support and facilitate the automatic interaction between semantic services and service consumers (Akkiraju *et al.*, 2005).

According to Stollberg, Hepp, and Fensel (2010:14), WSDL-S partly realizes the semantically annotated services. Hence, it is considered a lightweight annotation framework. In this sense, WSDL-S might not necessarily be an appropriate choice for real-world service annotations. Moreover, it is tightly-coupled to WSDL described services, and would need major updates for services described in other languages.

4.3.2. SAWSDL

The Semantic Annotations for Web Services Description Language and XML Schema (SAWSDL) (Lausen, 2007; Lia, Abela & Scicluna, 2009) is a W3C recommendation service annotation approach that is almost similar to WSDL-S. It provides WSDL and XML schema extensions that could support the process of annotating multiple WSDL elements (Garcia-Sanchez *et al.*, 2009; Lia, Abela & Scicluna, 2009). In SAWSDL, WSDL elements are linked with ontological concepts using any semantic description language, as is the case with a WSDL-S mechanism (Stollberg, Hepp & Fensel, 2010).

SAWSDL is supported in WSMO through the use of tools, such as WSMO studio. However, SAWSDL is capable of handling few ontological concepts compared to what is provided for by WSMO. This could be a draw-back when dealing with complex services. In addition, according to a simple comparison of semantic description and annotation frameworks by Stollberg, Hepp and Fensel (2010:13), does not support ontological concepts that are highly expressive, which are essential for automation of different Web service aspects, such as service discovery.

The main difference between WSDL-S and SAWSDL is that SAWSDL does not provide support for annotating the sub-elements of the precondition construct in WSDL. In SAWSDL, preconditions and effects are not catered for, and this could limit service discovery and invocation to a keyword-based search (Stollberg, Hepp & Fensel, 2010:13). However, on the positive side, SAWSDL does support the discovery of services through categorization (Stollberg, Hepp & Fensel, 2010:16).

4.3.3. WSMO-Lite

WSMO-Lite is a WSMO inspired light-weight semantic annotation model building on SAWSDL. The main objective of WSMO-Lite is on supporting the seamless integration of intelligent services, by adding semantic annotations to existing WS technologies (e.g. WSDL) (Vitvar, Kopecky & Fensel, 2009). These annotations are expressed in RDF(S) (Kopecky & Vitvar, 2008). Furthermore, WSMO-Lite extends and supports SAWSDL with ontological annotations, functional annotations and non-functional annotations (Bussler *et al.*, 2004; Kopecky & Vitvar, 2008).

WSMO-Lite also simplifies the processes of annotating services, by opting for a bottom-up approach, where WSDL is the foundation, as compared to heavy-weight semantic description languages, such as WSMO and OWL-S that subscribe to a top-down approach (Kopecky & Vitvar, 2008). In WSMO and OWL-S, WSDL and SOAP, standards are not directly considered when semantically describing services. This means that WSMO and OWL-S can be applied to different types of services, such as RPC-based or RESTful, unlike WSMO-Lite, which is only aligned towards WSDL described services.

Although inspired by WSMO, WSMO-Lite is ontology-language independent - meaning that concepts used for annotations can be defined in any W3C language, based on RDF, such as OWL. WSMO-Lite does not subscribe to service descriptions completeness, as compared to WSMO. Moreover, it is not possible to define domain ontologies using WSMO-Lite. As a result, WSMO-Lite does not provide an environment where all aspects of services (e.g. ontologies, goals, and mediators) could be described semantically, as is the norm when using WSMO.

The main limitation of WSMO-Lite is that the behavioural aspects are not considered when annotating semantic services. However, this is catered for through the annotations of service-functional properties (Vitvar, Kopecky & Fensel, 2009). In addition, the expressiveness of WSMO-Lite depends entirely on the ontology language used. Although, this is not necessarily a limitation when languages, such as WSMO are used; it is a limitation when some languages are used, such as RDF(S), which are limited by design (Horrocks, 2008), and are less expressive.

4.4. SUMMARY

In this chapter, semantic models have been introduced. The focus is mainly on semantic descriptions and annotations. OWL-S and WSMO are presented as the two common semantic description models, whilst WSDL-S, SAWSDL, and WSMO-Lite, were presented as the common semantic annotation approaches within the field of the semantic Web services.

Based on the review of different semantic description and annotation models, WSMO and OWL-S were found to be highly expressive, yet with steep learning curves for developers. Nevertheless, OWL-S and WSMO models are not dependent on any service description language (e.g. WADL or WSDL). On the contrary, the light-weight semantic annotation approaches (e.g. WSDL-S) are more inclined towards WSDL-based services. In addition, these standards (i.e. light-weight approaches) provide limited expressivity, with regard to service ontologies. This could be a limitation in delivering Web services that could be autonomously and automatically processed and understood by machines.

In this chapter, a number of facilitation tools for WSMO and OWL-S were also briefly presented. These tools vary from a comprehensive toolset, such as WSMO studio to lean libraries, such as WSDL2WSMO translators. Nevertheless, most of these tools do not systematically, and by default, support the complete lifecycle of semantic services. They merely facilitate the process of building semantic descriptions, and simply annotating existing services. This means that uniform and rapid engineering of semantic services is not supported by these existing semantic-based tools; which is what this study attempts to address.

It is also noted in this chapter that some of these tools, such as WSMO Studio, are meant for expert developers, who are quite knowledgeable with the concepts of logical formalisms. Hence, it could be a challenge for average developers to use these tools for building their own semantic services. The overall objective of this study is that of having a uniform – and yet user-friendly – engineering environment that could support, simplify, and accelerate the process of engineering intelligent semantic services.

In Chapter 5, the fundamental building blocks that make up intelligent semantic services are identified, characterized, and grounded.

CHAPTER 5: IsS Definition and Basic Building Blocks

Chapter 5 formulates the proposed solution by providing an elaborative definition for the term intelligent semantic services. Additionally, the fundamental building blocks that make up the intelligent semantic service are discussed. In addition, software agents, which are considered for realizing the intelligence building block, are briefly discussed.

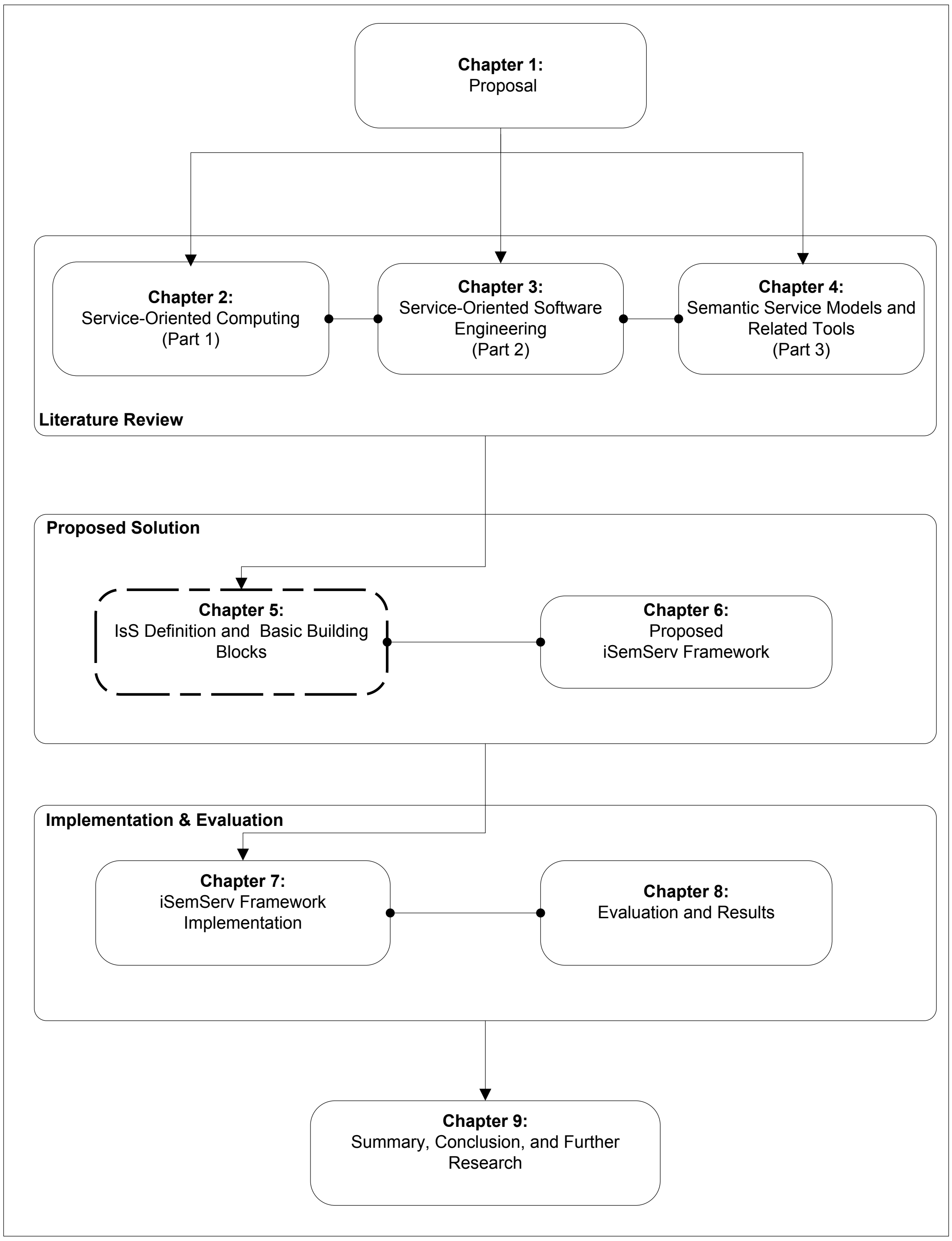

Chapter 1:
Proposal
Chapter 2:
Service-Oriented Computing
(Part 1)
Chapter 3:
Service-Oriented Software Engineering
(Part 2)
Chapter 4:
Semantic Service Models and Related Tools
(Part 3)
Literature Review
Proposed Solution
Chapter 5:
IsS Definition and Basic Building Blocks
Chapter 6:
Proposed iSemServ Framework
Implementation & Evaluation
Chapter 7:
iSemServ Framework Implementation
Chapter 8:
Evaluation and Results
Chapter 9:
Summary, Conclusion, and Further Research

Figure 5.1: Overall Thesis Structure

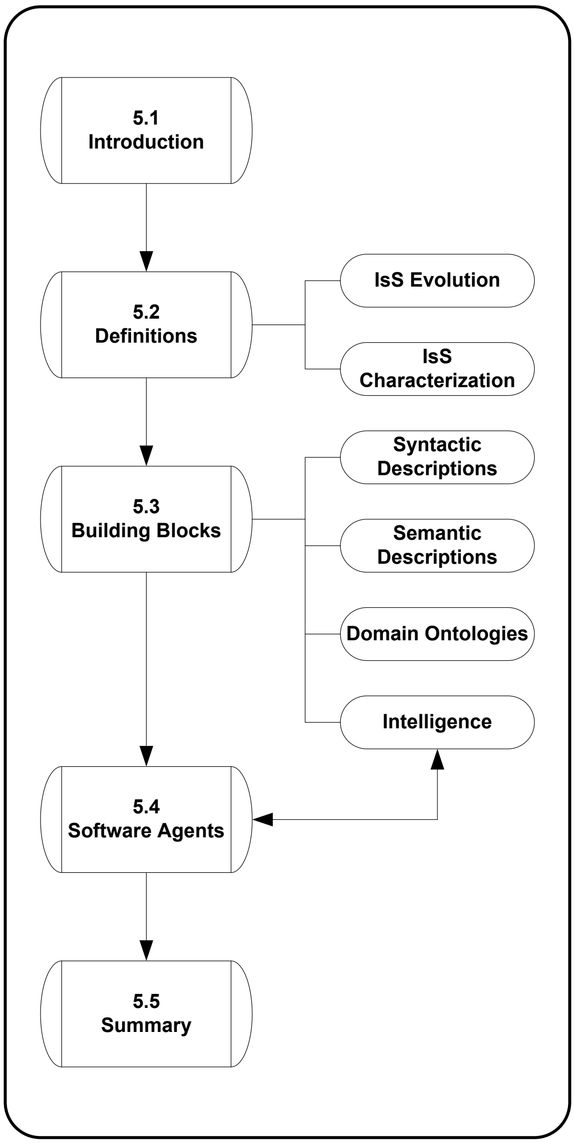


Figure 5.2: Chapter 5 Summary

5.1. INTRODUCTION

Semantic Web Services (SWS) are, in some cases, referred to as Intelligent Web Services (IWS), mainly because of the semantic descriptions in Web services (WS) that could be queried and interpreted by software agents (Balzer, Liebig & Wagner, 2004; Feier *et al.*, 2005; Gomez-Perez & Euzenat, 2005; Guha, 2009; Wang *et al.*, 2007). In some instances, the integration of software agents, Web services, and ontologies is considered by other researchers as the basis of intelligent services (Garcia-Sanchez *et al.*, 2009; Kanellopoulos & Kotsiantis, 2006; Lewis, 2008; Papazoglou, 2001; Simula, 2007; Soe-Tsyr & Kwei-Jay, 2003).

The Web Services Architecture (WSA) document (Booth *et al.*, 2004), states that:

"A Web service is an abstract notion that must be implemented by a concrete agent. The agent is the concrete piece of software or hardware that sends and receives messages, while the service is the resource characterized by the abstract set of functionality that is provided."

The definition of Booth *et al.* (2004) implies that Web services can be regarded as agents or parts of agent systems. Zhu and Shan (2005) also state that *"WS can be regarded as part of agent systems"*. In addition, the Semantic Web Services Architecture[32](SWSA) attempts to enable a high degree of automation in semantic services, by addressing all processes of Web services (e.g. discovery, composition, invocation, etc.) through the use of software agents (Burstein *et al.*, 2005; Gümüs *et al.*, 2007). In SWSA, software agents are intended to provide and consume semantic services in an intelligent manner (Gürcan *et al.*, 2007).

Nevertheless, *intelligent semantic services (IsS)* are not clearly defined. Hence, in this chapter, the objective is to elaborate as to *what an intelligent semantic service (IsS) is* in the context of this study; *how it is distinct from traditional WS*, *SWS, and agent-based software systems*; and *what appropriate building blocks make up an IsS*.

The remainder of this chapter is structured as follows: Section 5.2 provides a clear and an elaborative definition of what is meant by the term *intelligent semantic service*

[32] See http://www.swsi.org for a detailed architecture

in the context of this study. Section 5.3 identifies and discusses the basic building blocks that are found to be essential to the formal characterization and realization of intelligent semantic services. In Section 5.4, software agents, with the particular focus on intelligent agents, which form part of the *IsS* building blocks, are discussed from the literature review perspective; and a motivation is also presented on why software agents are suitable for realizing intelligence in SWS.

5.2. DEFINITION

In this research, an *intelligent semantic service (IsS)* is meant to extend and leverage WS and SWS, with the intelligence implemented using intelligent agents. This is done to enable the emergence of services on the Web that are autonomous and automatable.

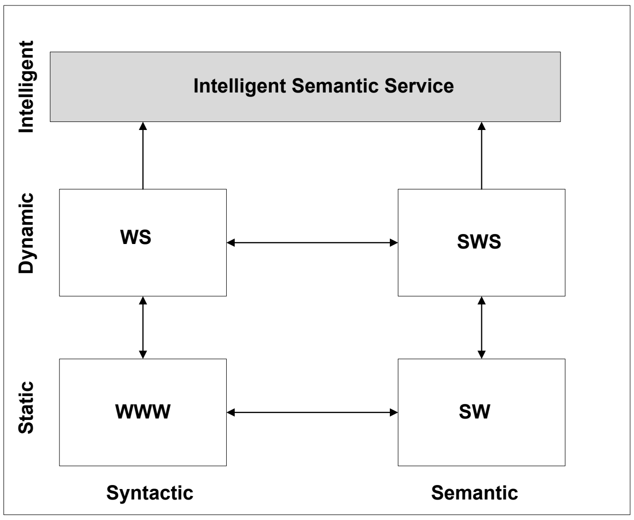


Figure 5.3: Intelligent Semantic Service Evolution

Figure 5.3 depicts an abstract position for an *IsS* in the context of the evolution of Web services. As may be noted, *IsS* inherits the properties and features of Web services and semantic services. Thus, in defining the term *intelligent semantic service*, we focus on two core notions, namely *semantics (i.e. semantic descriptions)* and *intelligence*, as depicted in Figure 5.4. These two concepts are essential towards the practical development of intelligent semantic services.

Generally, semantic descriptions enable services to be machine-processable and interpretable through formal specifications of functional, non-functional, and behavioural aspects of Web services (de Bruijn *et al.*, 2008:20-21). In addition, intelligence in the context of artificial intelligence, is commonly associated with autonomy, reactive, proactive, and collaborative or social ability properties (Protogeros, 2008). It is viewed as an approach of incorporating cognitive abilities into machines (e.g. software agents).

Henceforth, adopting the common intelligence properties found in (Jennings & Wooldridge, 1998; Protogeros, 2008) and semantic Web key enablers found in (de Bruijn *et al.*, 2008; Studer, Grimm & Abecker, 2007) and based on the objectives of this research, an *intelligent semantic service (IsS)* is defined *as a semantically enabled software unit representing some business functionality that could be accessed through the Web, and is capable of being: (1) autonomous, (2) proactive, (3) reactive (4) machine-processable and understandable, as well as (5) collaborative.*

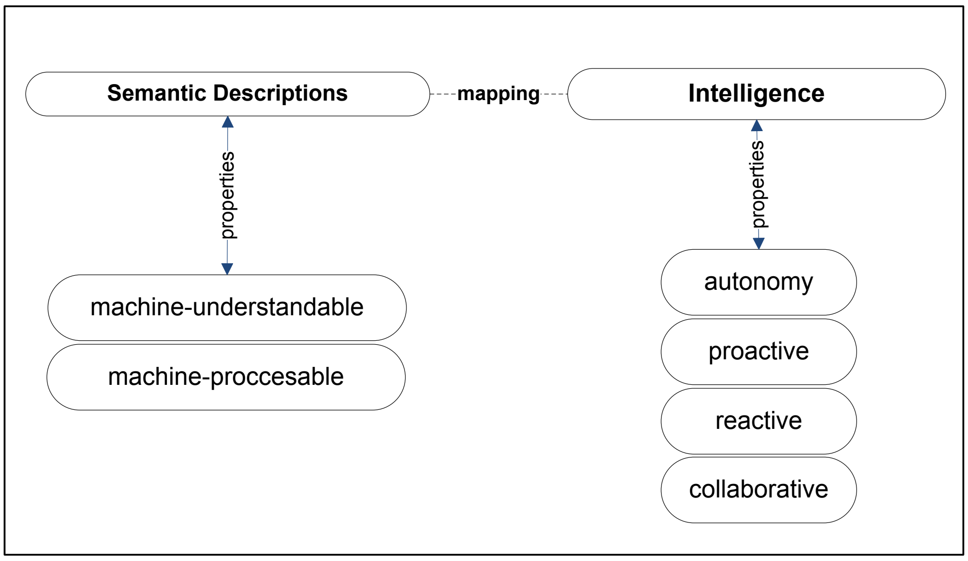


Figure 5.4: IsS Definition Properties

The properties in the definition are further expanded as follows:

- ***Semantically-enabled:*** This refers to the enrichment of services with semantic descriptions, as derived from domain and service ontologies created using semantic models, such as WSMO and OWL-S. This property enables a developed *IsS* to be machine processable and understandable in a manner that service aspects, such as discovery, selection, and composition, could be automated.

- ***Autonomous:*** This property characterizes an *IsS* as a service that has the ability to act on behalf of its owner, and to carry out the required actions with limited or no human intervention (Jennings & Wooldridge, 1996).
- ***Proactive:*** This refers to the ability of the service to show goal-directed behaviour and to be able to take initiatives where necessary (Protogeros, 2008; Zhu & Shan, 2005).
- ***Reactive*:** This refers to the ability of a service to react to its environment and situation. This further allows the service to adjust its behaviour, based on situational circumstances of the service consumer, service requesting device, service embedded behaviour, service boundaries, and current location. This property could also be referred to as context-awareness.
- ***Collaborative:*** The collaborative property enables an *IsS* to be able to communicate openly and seamlessly with other services available for consumption by the Web community. In other publications, this property is referred to as the social ability (Zhu & Shan, 2005).

Some of the properties in the *IsS* definition have been widely used in Artificial Intelligence (AI), especially intelligent agents (Jennings & Wooldridge, 1996; Protogeros, 2008). Thus, in our opinion, the concept of intelligent agents cannot be ignored when considering intelligent semantic services. Herein, agents are also discussed in this chapter, in order to highlight the fundamental mappings between intelligent agents and semantic services (Usman *et al.*, 2006).

The following section discusses the fundamental building blocks that constitute an *IsS*.

5.3. FUNDAMENTAL BUILDING BLOCKS

The fundamental building blocks that comprise an *IsS* are presented in Figure 5.5. These building blocks demonstrate that an *IsS* is a non-atomic unit. However, in order to achieve a degree of intelligence, the building blocks are interconnected to form an *intelligent semantic service*. These building blocks were conceived through a literature review of Web services, Ontologies, Semantic Web Services, and Intelligent Agents, as was discussed throughout the background chapters.

The building blocks are directly related to the *IsS* definition and its properties. The first three building blocks are linked to the *semantically-enabled* property, whilst the *intelligence* building block is linked to the rest of the other properties, such as autonomy. Additionally, these building blocks guide the process on how an *IsS* could be simply engineered. Furthermore, they (i.e. building blocks) enable us to decide on the engineering methodology to follow when designing, modelling, and implementing *intelligent semantic services.*

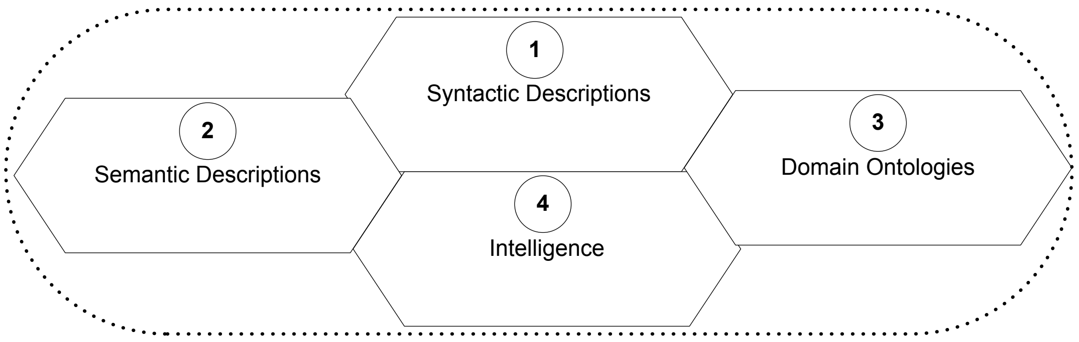


Figure 5.5: IsS Basic Building Blocks

In the following sub-sections, we shall describe each building block in detail.

5.3.1. Syntactic Descriptions

The main goal of service providers is to satisfy a business need; that is, services provided need to be of value to both the provider and consumers (Cardoso, Voigt & Winkler, 2008). Because of continuous changing business and user requirements in the service economy, services provided on the Web cannot be rigid; and therefore, they need to be captured in a manner that promotes interoperability, and adaptation.

To achieve this, various services need to be specified, represented, and described, using standard approaches.

Thus, one of the main building blocks that have been identified as core to the formation of an *IsS* is the *syntactic descriptions,* also referred to as “service descriptions”. *IsS* can only be beneficial to providers and consumers, if it captures and facilitates the delivery of some valuable business services; and this value is described and discoverable.

The *syntactic descriptions* building block captures the overall syntactic, non-functional, functional, and behavioural properties of an *IsS;* as it is the norm with traditional WS and SWS.

5.3.2. Semantic Descriptions

As defined, ontologies are a “formal specification of shared conceptualization” (Gruber, 1993). This means that ontologies capture unambiguous accepted terminologies representing a particular domain, which can be commonly interpreted, and processed by humans and software programs (Stollberg, 2006). Therefore, ontologies are essential in the development of intelligent semantic services.

However, ontologies differ in categorization (Studer, Grimm & Abecker, 2007:78). They can be grouped, according to top-level ontology, domain ontology, service ontology, and application ontology. The main groups that have been identified as important to the composition of an *IsS* are the domain and service ontologies. Domain ontologies capture domain-specific knowledge, and service ontologies[33] capture knowledge about a specific service operating within the boundaries of a particular domain. As a result, the other main building block identified for an *IsS* is the *service ontology or semantic descriptions,* as depicted in Figure 5.5.

Semantic descriptions are useful for semantically describing service inputs, outputs, pre-conditions, effects, transport messages, non-functional properties, and service processes – using the concepts defined in the domain ontology. The *semantic descriptions* building block’s main goal is to enrich Web services with semantic knowledge.

[33] Service ontologies are referred to as semantic descriptions in the context of Semantic Web Services.

5.3.3. Domain ontologies

In order for Web services to efficiently interoperate with one another in an automated and intelligent manner, domain ontologies are essential. These are necessary, in order to ensure that services share the same knowledge and understanding about a particular domain. As one of the building blocks for the formation of an *IsS*, *domain ontologies* provide shared vocabularies that are service-independent, and are the basis of semantic descriptions.

In a number of domains, universal and commonly agreed-upon ontologies already exist, such as in medicine, tourism, and transport (Studer, Grimm & Abecker, 2007:78). These ontologies can be re-used, rather than developed anew for different tasks in SWS. In our work, semantic services could use domain-related knowledge to “share the same interpretation of concepts and terms” (Garcia-Sanchez *et al.*, 2009) during the phases of service discovery, selection, composition, invocation, and execution.

5.3.4. Intelligence

Software agents have been linked to the realization of SWS by various authors (Garcia-Sanchez *et al.*, 2009; Hendler, 2001; Lewis, 2008). However, few pilot applications that demonstrate their practicality and viability in semantic services exist. In our work, the *intelligence* building block is considered for: (1) Defining and realizing the intelligent behaviour, according to the *IsS* properties; (2) facilitating the reasoning over and processing of domain and service ontologies; (3) minimizing human involvement in service requests and provisions; and (4) promoting a high degree of automation in various service processes (e.g. discovery and, selection).

The *intelligence* building block is based on the properties of intelligent agents, as described in Section 5.4. The agent itself is not a service, but is intended to facilitate the activities involved in delivering intelligent semantic services. In other words, an intelligent agent is chosen to leverage an *IsS* with intelligence capabilities.

5.4. SOFTWARE AGENTS

According to Usman *et al.* (2006), software agents and Web services share a number of commonalities that could address the challenges hindering the intelligence of services on the Web. This is further highlighted by Garcia-Sanchez *et al.* (2009), who advocate that service intelligence could be widely realized through the combination of software agents, WS, and SWS. In principle, agents and SWS can complement each other in a number of ways. For instance, by integrating them at different levels, in order to minimize the challenges faced by each technology when applied and deployed independently.

For example, software agents could be very useful in dynamically co-ordinating service requests and provisions in a distributed environment. In addition, WS and SWS could be useful in facilitating the collaboration between multiple heterogeneous agents.

However, software agents suffer from a number of challenges when it comes to collaborations outside their own domain. One of the challenges is the use of closed and platform-dependent communication protocols, such as Remote Method Invocation (RMI) and Internet Inter-ORB Protocol (IIOP), which makes it a challenge to achieve interoperability when agents are implemented without the support of open standards (García-Sánchez *et al.*, 2011; Schaaf & Maurer, 2001). Contrarily, WS and SWS operate passively when implemented, independent of agent systems. Software agents are active entities, and integrating them with Web services, which employ open standards for messaging, could promote the realization of intelligent semantic services.
In essence, a *software agent* can be defined as a computational entity that is capable of accomplishing users' tasks in an autonomous manner (Biermann, 2004; Garcia-Sanchez *et al.*, 2009; Jennings & Wooldridge, 1996; Protogeros, 2008).This means that it is capable of acting on its own, without much human involvement. An agent is usually situated in an environment, where it is capable of interacting with other agents, for the purpose of completing a given task.

There are different types of software agents, as illustrated in Figure 5.6. Furthermore, agents are classified, according to various properties, as exemplified in Figure 5.7. These types of agents are identified and discussed in detail by Nwana (1996). For example, collaborative, mobile, interface, smart, information, reactive, and hybrid agents.

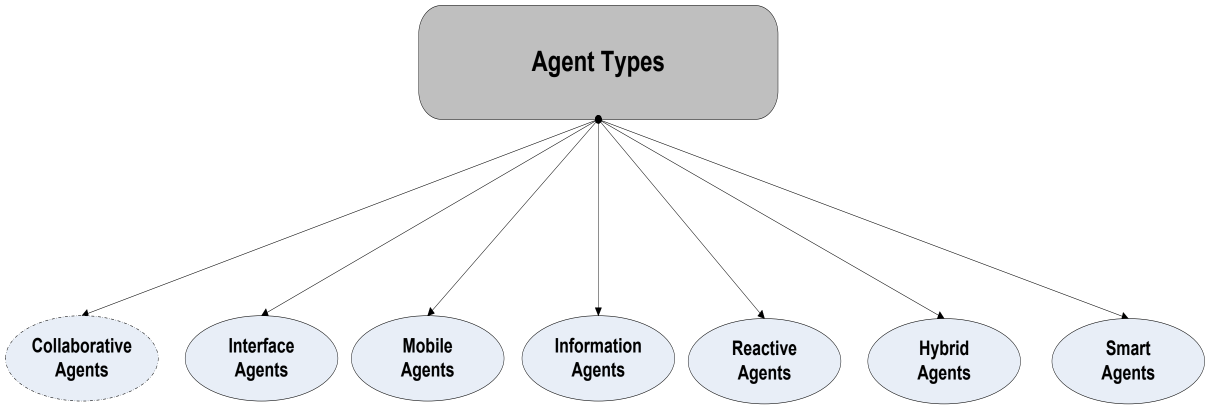


Figure 5.6: Typology of Agents

As depicted in Figure 5.6, core to this thesis are the *collaborative agents*; which are capable of acting co-operatively, rationally, autonomously, and have the ability to learn(Nwana, 1996; Protogeros, 2008) in their respective environment, as depicted in Figure 5.7. Furthermore, *collaborative agents* possess social abilities. This means that collaborative agents are also able to communicate with other agents in an "open and time-constrained multi-agent environment" (Nwana, 1996). Collaborative agents also have learning and reasoning capabilities, but these are not core to their ultimate operations (Nwana, 1996).

In the context of this thesis, *collaborative agents* are considered to be having similar capabilities as *intelligent agents*, particularly with regard to autonomy and reasoning.

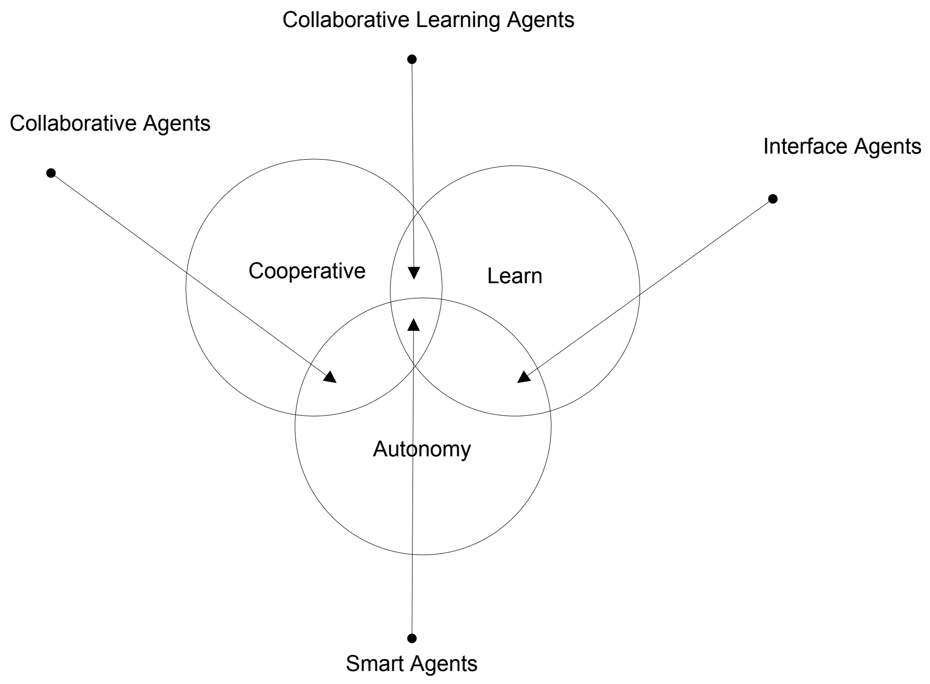


Figure 5.7: Classification of Agents

In this regard, intelligent agents are fundamental to the realization of the *intelligence building block,* as discussed in Section 5.3.4. As defined by Jennings and Wooldridge (1998), intelligent agents are computational entities *“capable of flexible autonomous actions, in order to meet design objectives”.* Accordingly, intelligent agents are appropriate for implementing intelligent semantic services, as they partially capture the necessary *IsS* properties. As already stated, collaborative agents also have learning and reasoning capabilities, but these are not core to their ultimate operations (Nwana, 1996).

The agents’ properties that are of importance, and are captured by the incorporation of intelligent agents are: autonomy; collaborative (i.e. social ability); reactivity; and pro-activeness. Additional properties are: rational; extensibility; and situational-awareness. These are broadly captured within the semantic and syntactic descriptions.

According to Blois, Escobar, and Choren (2007); intelligent agents can be useful for the development of SWS products, and according to Jennings and Wooldridge (1998), software agents are capable of realizing a number of processes within a service engineering domain, such as dynamic service discovery, composition, and invocation. Consequently, intelligent agents have been implemented in real-world software applications for different purposes (Garcia-Sanchez *et al.*, 2009; Jennings & Wooldridge, 1998; Nwana, 1996; Protogeros, 2008), such as:

- Workflow management;
- Dynamic information retrieval and management;
- Interoperating legacy systems;
- Solving inherently distributed problems (e.g. telecommunications network management); and
- Enhancing systems' modularity, speed, reliability, flexibility, re-usability, and reducing complexity.

Further details, such as detailed agents' properties, different agent architectures, communication approaches, languages and transport mechanisms are beyond the scope of this thesis, and as such are not covered. However, for proof of concept implementation, a common agent architecture, called the Java Agent Development framework (JADE) (Protogeros, 2008)shall be briefly covered in Chapter 8.

5.5. SUMMARY

Intelligent semantic services are emerging as an attempt to address issues related to dynamic service-oriented systems; where heterogeneous Web services are composed, with the purpose of delivering integrated value-added services. There are still a number of challenges that need to be addressed, in order to achieve intelligent services on the Web. However, as a base, Web services can be augmented with semantic descriptions and intelligence, as derived from software agents – to achieve automation and autonomy in business services.

The main focus of this chapter was to define the term "intelligent semantic service", which has been coined, based on the foundations of semantic web services and software agents. Furthermore, since the main objective of our work is toward simplifying and accelerating the process of engineering intelligent semantic services, it was essential that that we also determine, define, and describe what constitutes an intelligent semantic service.

As a result, the basic building blocks for intelligent semantic services were identified and described. The building blocks were identified after a literature review and practical observations include syntactic descriptions, semantic descriptions, domain ontologies, and intelligence. These are meant to realize intelligent semantic services and minimize user interventions during service requests and provisioning on the Web.

The basic building blocks are essential for the overall process of engineering semantic services; and they are the basis of the proposed service creation framework, which will be presented in the next chapter.

Software agents were discussed as being of relevance to the scope of this study, and properties that are relevant to intelligent semantic services were also covered. In what follows, in Chapter 6, design principles that underpin the service creation framework will be determined and explained. The proposed conceptual service creation framework is also presented and explained. The methodology preferred for engineering intelligent semantic services is also described.

CHAPTER 6: Proposed iSemServ Framework

This chapter presents the proposed service creation framework to address the challenges discussed in Chapter 1. The discussion starts off with the design principles that are the basis for the proposed service creation framework. This is followed by a brief introduction of a model-driven engineering methodology that is adopted for engineering intelligent semantic services. The framework is presented in a multi-layered format, which represents the basic building blocks, as discussed in the previous chapter.

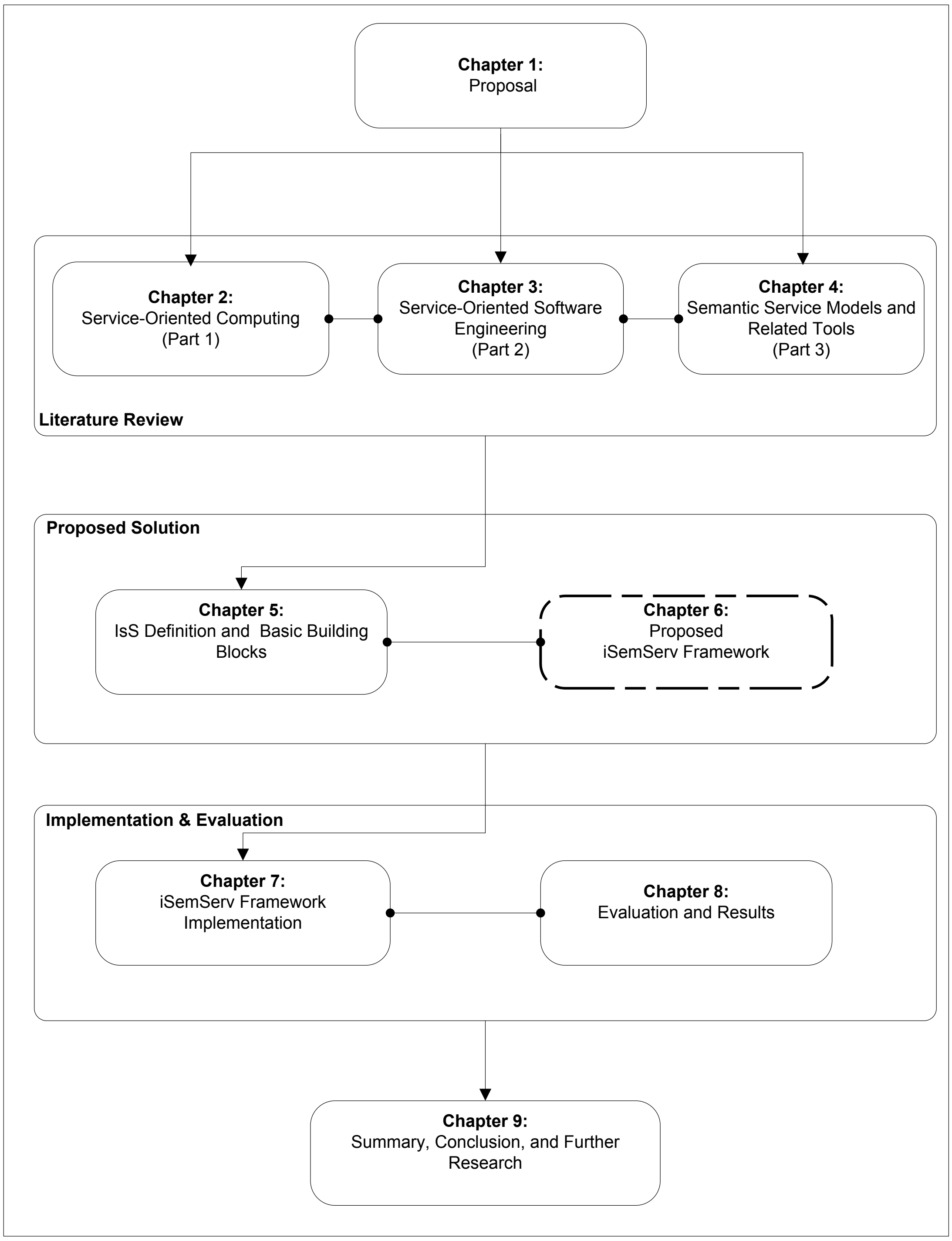

Chapter 1:
Proposal
Chapter 2:
Service-Oriented Computing
(Part 1)
Chapter 3:
Service-Oriented Software Engineering
(Part 2)
Chapter 4:
Semantic Service Models and Related Tools
(Part 3)
Literature Review
Proposed Solution
Chapter 5:
IsS Definition and Basic Building Blocks
Chapter 6:
Proposed
iSemServ Framework
Implementation & Evaluation
Chapter 7:
iSemServ Framework Implementation
Chapter 8:
Evaluation and Results
Chapter 9:
Summary, Conclusion, and Further Research

Figure 6.1: Overall Thesis Structure

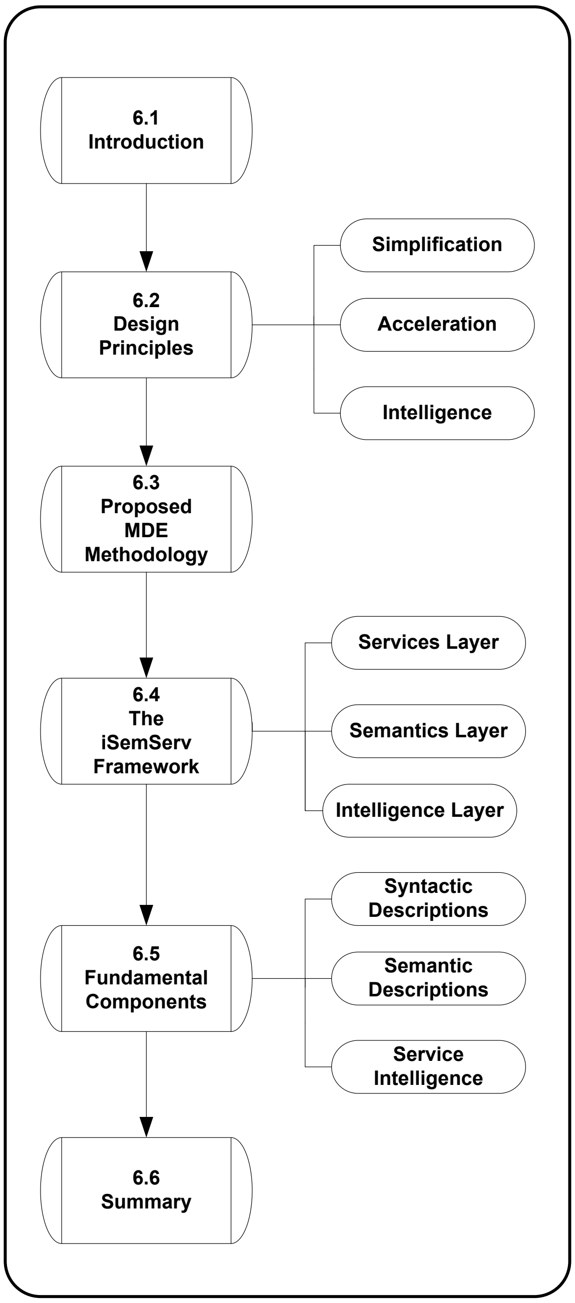


Figure 6.2: Chapter 6 Layout

6.1. INTRODUCTION

This chapter describes the service creation framework for *simplifying* and *accelerating* the process of engineering intelligent semantic services (*IsS*). The framework is termed ***i*SemServ –** from the word ***i***ntelligent ***Sem***antic ***Serv***ices.

The concepts described in the previous chapters, especially in Chapters 2 – 5, and the challenges explained in Chapter 1, have provided us with an understanding of the current state of affairs within the Web services (WS), Semantic Web (SW), Semantic Web Services (SWS), Ontologies, and Intelligent Agent domains. This was essential for grounding the challenges highlighted in this thesis, and on what has already been proposed within the research environments of the domains listed above.

Thus, based on the literature review of the related work (cf. Chapters 1 – 4), a number of challenges and shortcomings of existing semantic service engineering models were identified. This has motivated us to propose a unified service-creation framework, which aims to address some of the identified challenges.

Concisely put, the shortcomings that were identified include:

- The lack of methods and tools for simplifying and accelerating the process of engineering intelligent semantic services within a unified environment (cf. Chapter 1).
- The lack of standardized methods for formulating semantic service descriptions and annotations (cf. Chapters 1 - 3).
- The lack of efficient interoperability between different semantic services models (e.g. WSMO & OWL-S) and languages (e.g. OWL & WSML) (cf. Chapter 4).
- The use of low-level lexical notation semantic languages (e.g. WSML), thus leading to a steep learning curve for developers.
- Incompatible and disconnected tools for developing semantic services (cf. Chapter 4).

- Current solutions are tightly coupled to specific semantic service models and ontology languages. For example: OWL-S IDE is tightly coupled to OWL-S and WSMO Studio is tightly coupled to WSMO (Dimitrov *et al.*, 2007; Srinivasan, Paolucci & Sycara, 2006), leading to restrictive development environments.
- The lack of support for building intelligent semantic services within existing semantic service environments.
- The non-integration of existing semantic technologies with mature Web service technologies.

The remainder of this chapter is structured as follows: In order to address some of the challenges listed above, essential design principles for the formulation of the proposed service creation framework are presented in Section 6.2. Section 6.3 discusses the service engineering methodology for the proposed service creation framework; and this is based on the literature review presented in Chapter 3. The *i*SemServ framework is presented and described in Section 6.4. We elaborate on the key components from the *i*SemServ framework in Section 6.5.These are: syntactic descriptions, semantic descriptions, and intelligence wrapping. The chapter is concluded with a summary in Section 6.6.

6.2. DESIGN PRINCIPLES

Semantic service technologies generally adhere to the enabling standards of the Semantic Web as discussed in Chapter 2 (Section 2.3). In addition to the Semantic Web enabling standards, for the proposed service creation framework, the following design principles were identified. The principles discussed in Section 6.2.1 - 6.2.3 were identified through are view of related work based on the challenges that the *i*SemServ framework addresses.

The *i*SemServ framework needs to address the design principles, as illustrated in Figure 6.3. The design principles are grouped into three main categories, namely; *simplification, acceleration*, and *intelligence*. These categories emanate from the core focus of this thesis; that is, *simplifying* and *accelerating* the process of engineering *intelligent* semantic services.

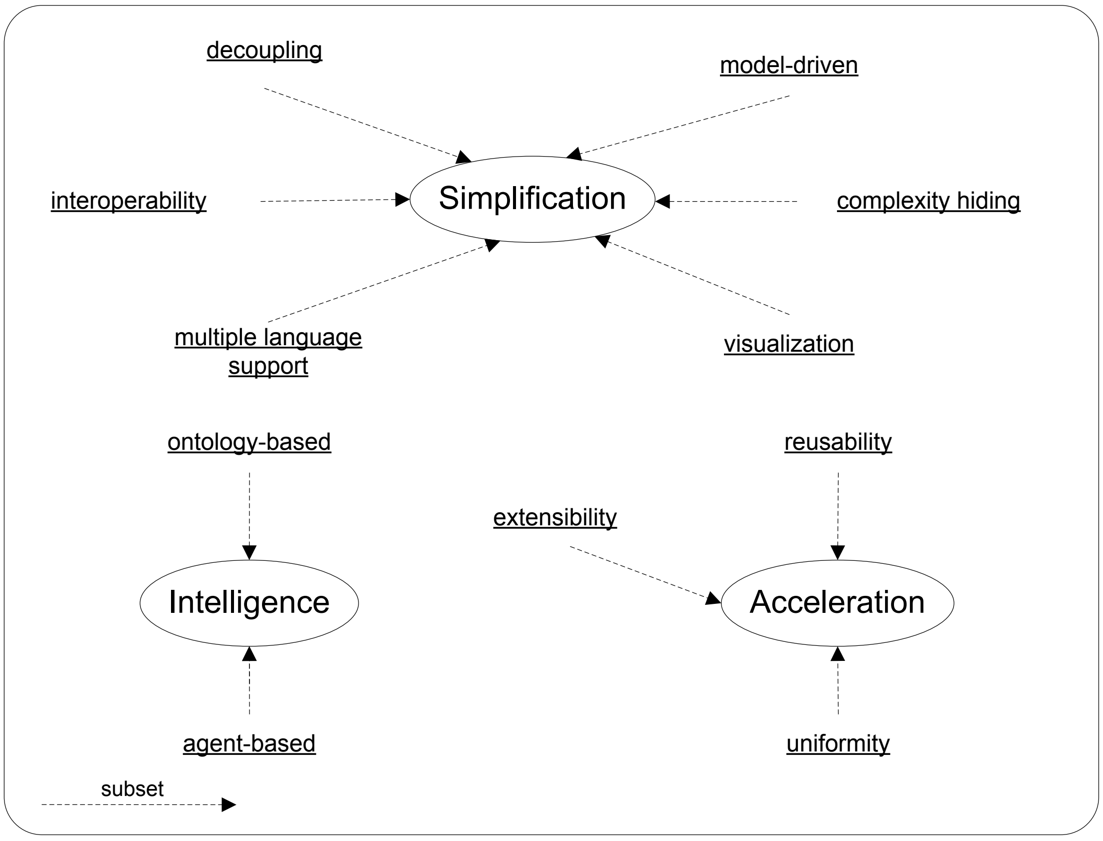


Figure 6.3: *i*SemServ Design Principles

6.2.1. Simplification

- *Model-driven* – In order to simplify the semantic service development experience, the *i*SemServ framework follows a model-driven approach rather than a code-driven approach. A model-driven approach is considered as efficient and exhaustive when it comes to developing software systems; and this is also true for service-based systems (Srinivasan, Paolucci & Sycara, 2006). Models are also important for code generation, due to the different levels of abstraction (Nassar *et al.*, 2009); thus maximizing complexity hiding and service engineering productivity.

- *Decoupling* – This principle requires that the *i*SemServ framework promotes the separation of concerns (SOC).Similar to WSMO elements (de Bruijn *et al.*, 2005a), the framework needs to support the definition of syntactic descriptions, semantic descriptions, and intelligence in an independent manner (Erl, 2008). However, these components need to be aware of each other, and to be easily integrated when needed.

- *Multiple Language* – Existing frameworks tend to only accommodate one particular language for describing services syntactically and semantically. Those that claim to support language independence, such as the ODE-SWS framework, tend to simply focus on semantic annotations; and they do not address the challenges of service descriptions and intelligence (Corcho *et al.*, 2003). In the proposed approach, the *multiple language support* requirement enables the framework to support different semantic description and syntactic description languages.

- *Complexity hiding* – The framework needs to support approaches that could aid service developers in rapidly implementing service components and re-using existing ontologies. In addition, it is necessary to support the use of tools that are capable of reducing complexities associated with intelligent semantic services development.

- *Interoperability* – As noted for the *decoupling* principle, the proposed framework needs to accommodate the implementation of different components (e.g. semantic descriptions) independently; but at the same time, the components need to be aware of, and to interoperate with one another. Interoperability is core to service-oriented environments. Thus, our framework needs to support the interoperability of different components and services.

- *Visualization* – The development of semantic descriptions is quite a complex and resource intensive process. Thus, without the availability of graphical and user-friendly tools, it can be daunting and error-prone to produce reliable semantic descriptions. Hence, the framework proposed in this thesis needs to embrace components that support the visualization of domain ontologies and semantic descriptions. This approach is also used in a few existing semantic tools for designing and viewing ontologies, such as in the Web Services Modelling Toolkit (WSMT), where *WSMOViz (Kerrigan, 2006)* is implemented – to guide the developer visually through the process of creating and editing ontologies (Kerrigan *et al.*, 2007).

In our case, the visualization requirement is one prerequisite for dealing with the issues of complexity hiding, especially when augmenting generated semantic descriptions and intelligence.

6.2.2. Acceleration

- *Uniformity* – The developers of intelligent semantic services need to be able to uniformly and cohesively perform all the activities (e.g. service modelling, development, description, annotation, and intelligence wrapping) of building semantic services within one environment. The framework needs to facilitate the engineering of services in a unified manner, in order to minimize the service development time and costs.

- *Extensibility* – The components of the framework need to have the ability to be extended to include new functionalities, when needed.

- *Reusability* – The components of the *i*SemServ framework need to be re-usable. For instance, semantic descriptions defined within the proposed framework should be available for re-use within other semantic services' environments. This is essential for the purpose of promoting re-usability and easier integration of technologies within the service-based ecosystem (Agarwal *et al.*, 2005). This could further lead to lower service development costs and prompter service deployment and publication.

6.2.3. Intelligence

- *Ontology-based* – The *i*SemServ framework deals with two aspects to realize intelligence in services, namely: ontologies and agents. Thus, one of the core requirements to realize an intelligent semantic service is an ontology-based framework. Similar to WSMO (de Bruijn *et al.*, 2005a) design requirements, services produced using the *i*SemServ need to be ontology-based, particularly for realizing the semantically rich principle embedded within the *IsS* definition.

- *Agent-based* –Software agents are used regularly in solving complex software problems, and are also appropriate for exposing and consuming Web services (Greenwood & Calisti, 2004). In the *i*SemServ framework, the agent-based design principle is essential for introducing the notion of intelligence into semantic services in concurrence with the ontologies realized through the ontology-based requirement. This is essential in realizing some of the promises of SWS, such as automatic service discovery.

6.3. PROPOSED MDE METHODOLOGY

In the software and Web engineering domains, there are a number of mature methodologies that could be used to guide the development of semantic services. For instance, techniques such as Web Modelling Language (WebML) (Brambilla *et al.*, 2006), Unified Modelling Language for Services (UML-S) (Dumez *et al.*, 2008), XML, and business process modelling languages, such as BPMN (White, 2004) have been used extensively; and in some cases, they have been adopted to model Web services, Web applications, and complex end-to-end enterprise service-based solutions (Sun *et al.*, 2009). Some of these techniques are grounded in the Model-driven Architecture (MDA), an initiative of the Object Management Group (OMG) (Sun *et al.*, 2009), which is appropriate for modelling services in a technology platform-independent manner.

MDA deals with the complete life cycle of designing, deploying, integrating, and managing systems and services, using associated OMG open standards, such as UML (OMG, 2010b). It focuses on using high-level platform-independent models (PIMs) in designing systems and services. Furthermore, it handles the automatic transformation of these PIMs into platform-specific models (PSMs), and the generation of programming code stubs for the transformed PSMs (OMG, 2010b;

Xiaofeng *et al.*, 2006). MDA intrinsic features are those of promoting strict decoupling and isolating system and business logic from the technology platform used to implement the actual system (Sun *et al.*, 2009).

To design and realize the *i*SemServ framework that satisfies the design principles described in the previous section, a MDA, primarily the model-driven engineering (MDE) (Qafmolla & Cuong, 2010) methodology, is employed for engineering intelligent semantic services. The MDE methodology is chosen because of its essential benefits, such as the isolation of application logic and implementation technology, and support for simplifying and accelerating the design of systems and services (Sun *et al.*, 2009) without depending on a particular platform. In addition, other benefits that make MDA an appropriate choice for our *I*sS engineering approach include (OMG, 2010a):

- Reduced development time for new services;
- Improved service quality; and
- Rapid inclusion of emerging technology benefits into their existing systems.

The steps enumerated in Figure 6.4 illustrate the suggested MDE methodology for simplifying and accelerating the process of engineering intelligent semantic services. It should be noted that the methodology suggested presents alternative paths as illustrated by duplicate numbering. For instance, the starting point could either be the *service functional design* process (Step 1a) or the importing of existing *service models* (Step 2b). The outcome of the first step is a service model(s). It must be noted that step (1a or 1b) is preceded by the *service requirements* module, which does not directly form part of the proposed solution.

OMG recommends that models need to be defined using associated OMG modelling standards (Qafmolla & Cuong, 2010). Henceforth, for our suggested MDE methodology, the Unified Modelling Language (UML) is used for designing service models. UML is widely used across a number of organizations on account of its openness and extensibility features (Xiaofeng *et al.*, 2006). In the suggested methodology, new UML-compliant models can be designed, using the existing UML tools; or alternatively, existing models could be imported from different sources, as depicted in Figure 6.4 (Step 1b).

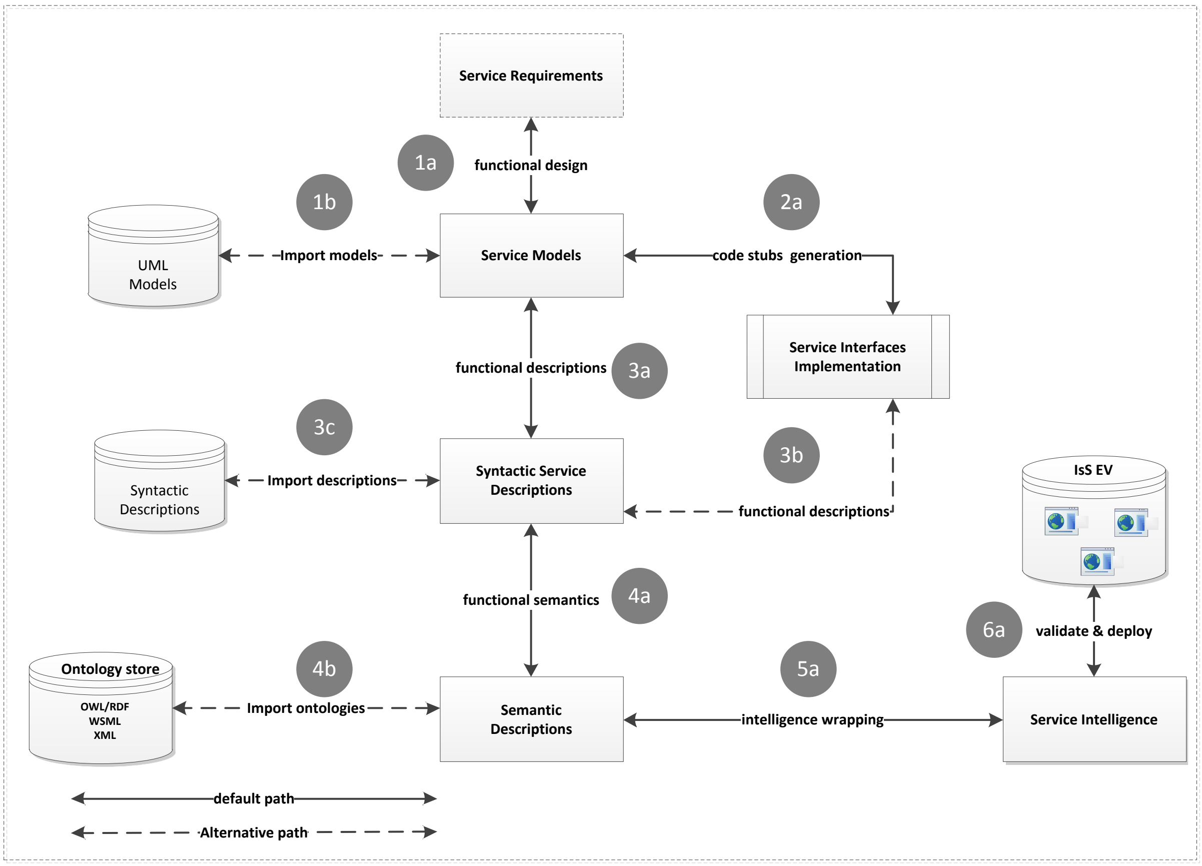


Figure 6.4: *i*SemServ Model-driven Engineering Approach

The next phase (Step 2a) involves the automatic transformation of the service model(s) into partial service logic code. The generated code stubs can be in a number of programming languages, such as Java or C#. Step 3a deals with automatically transforming service models into *syntactic descriptions,* such as WADL or WSDL (Christensen *et al.*, 2001; Filho & Ferreira, 2009). However, existing service descriptions could also be imported, as indicated in Step 3c.

Depending on the preferences of the developer, service descriptions could also be generated using the service logic code stubs generated and augmented in Step 2a. This particular phase is illustrated by Step 3b. There are a few tools available, such as Java2WSDL (Studer, Grimm & Abecker, 2007) that support the generation of service descriptions from service logic source code. Step 4a involves transforming validated syntactic service descriptions into semantic descriptions of choice, such as those prescribed by WSMO or OWL-S. It should be noted that these descriptions could either be derived from the service models designed in Step 1a, from the syntactic descriptions realized in Step 3, or from existing domain ontologies, as indicated in Step 4b. This is done in line with the service implementation for the purposes of annotating syntactic services with rich knowledge, using domain and service ontologies from disparate sources.

Once Web services have been semantically described using internal or external semantic descriptions; *service intelligence wrapping* follows in Step 5a. As part of the design principles, the wrapping of intelligence to semantic services is essential for achieving the promises of semantic services, such as automated service discovery, invocation, and execution.

The final step (Step 6a) in the suggested model-driven methodology is concerned with validating and deploying intelligent semantic services to an environment where intelligent semantic services could be automatically discovered, selected, invoked, executed, and monitored by the consumers.

It should be noted that the methodology presented in this section directly informs the proposed *i*SemServ framework, including its multi-layers and high-level abstract modules, as presented in the following section.

6.4. THE *i*SEMSERV FRAMEWORK

The *i*SemServ framework adopts the MDE methodology, as discussed in Section 6.3, to model, specify, describe, annotate, and add appropriate intelligence into semantic services. The core components of the framework are the *syntactic descriptor, semantic descriptor*, and *intelligence generator and wrapper*. These components are mapped to the fundamental building blocks described in Chapter 5 (Section 5.3). It should be noted that the semantic descriptor addresses both the domain ontologies and semantic descriptions building blocks.

In Figure 6.5, the proposed service creation framework is illustrated. The framework description in terms of its functional aspects, including its modules, is discussed. The framework is presented as a multi-layered architecture, made up of three core layers, namely: *services layer*, *semantics layer*, and *intelligence layer*. The implementation and validation of this framework is discussed in Chapters 7 and 8.

The fundamental modules of each layer are discussed in the following sub-sections.

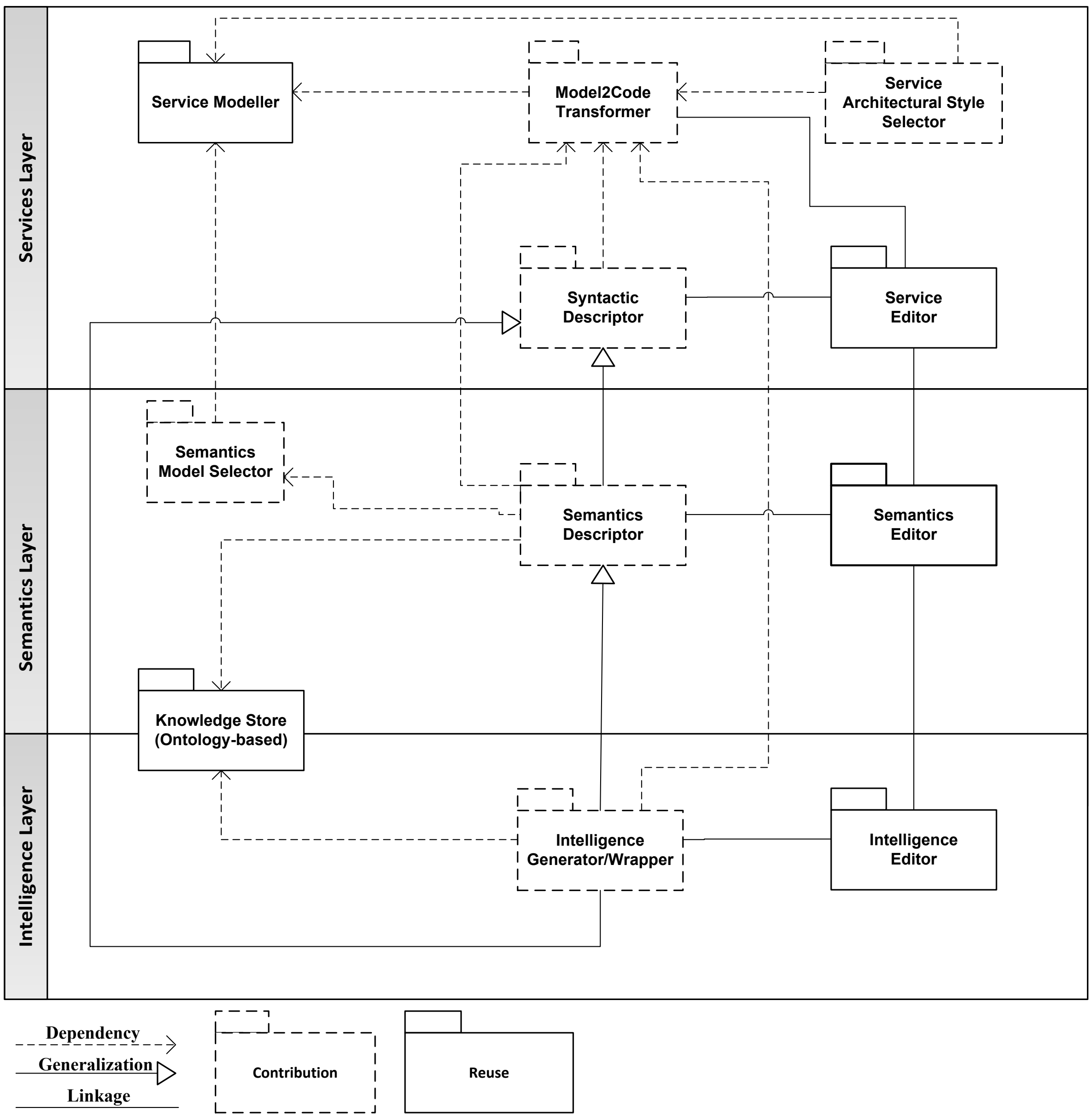


Figure 6.5: *i*SemServ Framework

6.4.1. Services Layer

In general, the service development process begins with a conceptualization of a service. That means the process begins at the requirements elicitation phase. Once all the activities to be performed by a concrete service are identified, *i*SemServ is employed to simplify and accelerate the process of building-up envisaged *IsS* or related applications.

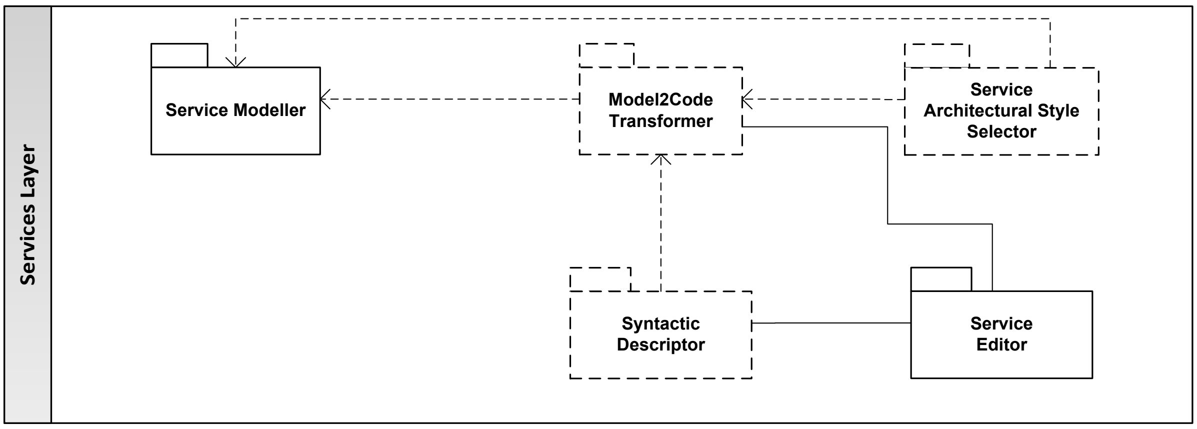


Figure 6.6: Services Layer Modules

In the services layer (cf. Figure 6.6), the initial step for engineering *IsS* deals with *service modelling,* facilitated by the *Service Modeller* module. *Service modelling* is mainly about representing envisaged *intelligent semantic services* using platform-independent models (PSMs).

This process is important, as it could simplify the service engineering process by enabling automatic generation of service code stubs from the model, thus reducing service development time and costs. In addition, this step promotes the decoupling of business logic and service logic.

Once a service model has been designed or imported, the model can be transformed automatically, using defined *Model2Code* templates and transformation rules into partial code stubs that represent the classes and operations relevant for *service logic* implementation. The code stubs could then be supplemented by the developer to realize a complete Web service, using any appropriate programming editor that is capable of interpreting the generated code could be used.

The programming editor as shown in Figure 6.6 is loosely linked[34] to the *Mode2Code transformer* and the *Syntactic Descriptor* - since they are responsible for generating the code that needs to be edited or validated. This is also done to promote uniformity when engineering semantic services.

As illustrated in Figure 6.6, the *i*SemServ framework supports the engineering of different types of services such as *SOAP (action-based) and REST (resource-based)* services. This is made possible by the *Service Modeller and Service Architectural Style Selector (SASS)* modules, which are some of the core contributions of this work. As may also be noted, the *SASS* module depends on the *Service Modeller,* in order to elicit and generate the type of services that the developer requires. For instance, the service could be REST-based, SOAP-based or both.

In the services layer, syntactic descriptions are automatically generated by the *Syntactic Descriptor* module, based on the service model realized or re-used in the *Service Modeller* module. The type of descriptions to be generated would depend on the annotations in the service model. For example, within the service model, the developer could annotate the models with WSDL stereotypes to indicate the preference to generate WSDL service descriptions. Once the syntactic descriptions are generated, and the service logic implemented, syntactic Web services are available for use; but they do not include semantic descriptions and intelligent features, as yet.

6.4.2. Semantics Layer

The semantics layer (e.g. *Semantic Descriptor module*) relies on the *Service Modeller* and *Model2Code transformer* modules for automatically generating domain ontologies and semantic descriptions. However, the syntactic descriptions produced in the services layer could also be used as input to the semantics layer's *Semantics Descriptor* module, as depicted in Figure 6.7. Since there are a number of semantic models that could be used to semantically describe Web services, our framework

[34] The link between the editors/validators and the relevant modules is loose mainly because only what is generated by the modules is visible to the editors/validators and not the modules. In other words: there is no operational relationship between the editors and associated modules.

provides the developer with the ability, through the use a defined UML profile[35], to choose the preferred semantics model(s) (e.g. WSMO or OWL-S). The *Semantics Model Selector* module would then be able to detect such a choice from the service model(s).

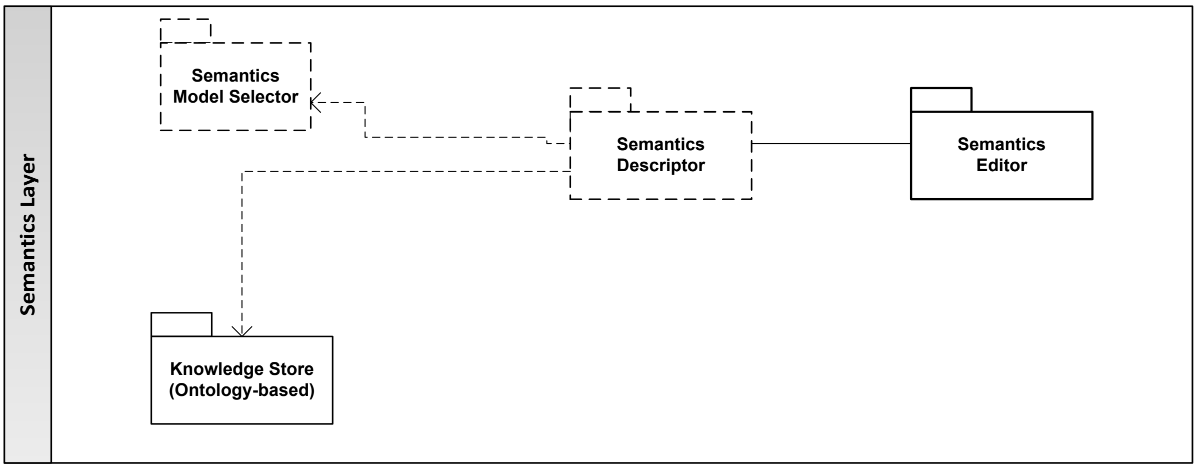


Figure 6.7: Semantics Layer Modules

Depending on the selection of a semantic model, partial semantic descriptions could then be automatically generated – using the *Model2Code* transformation rules, as defined in the services layer. Moreover, stand-alone tools, such as WSDL2OWL-S (Studer, Grimm & Abecker, 2007:313) and WSDL2WSMO (El Bouhissi, Malki & Bouchiha, 2008) could also be applied to realize semantic descriptions from syntactic descriptions.

The semantic descriptions generated by the suggested *Semantics Descriptor* module or existing translators are incomplete by default, the developer is then provided with a *Semantics Editor* module, in order to visualize, edit, augment, and validate the generated semantic descriptions.

Because the *i*SemServ framework relies on domain ontologies for semantic descriptions, an ontology-based *knowledge store* is provided, so that developers could also re-use existing ontologies to semantically describe services – where applicable. This *knowledge store* could be made up of domain and service ontologies. In addition, the knowledge store is shared across the intelligence layer,

[35] The *i*SemServ framework defined UML profile is presented in Chapter 7.

as depicted in Figure 6.5, for purposes of using the same knowledge to embed semantic services with intelligence.

The results out of the semantic layer are independent semantic descriptions and domain ontologies that describe services realized in the services layer. Because semantic descriptions are not automatable on their own; an intelligence component as described in the next section is needed to automatically process the semantic descriptions in intelligent manner that will lead to minimal user intervention during service consumption.

6.4.3. Intelligence Layer

At this stage, semantic services have been realized, and could be deployed to a semantic execution environment (e.g. WSMX) for automatic discovery and consumption. Nevertheless, in order to satisfy the principles that underpin the *IsS* definition, intelligence is wrapped into the semantic service(s) in this layer – to produce intelligent semantic services.

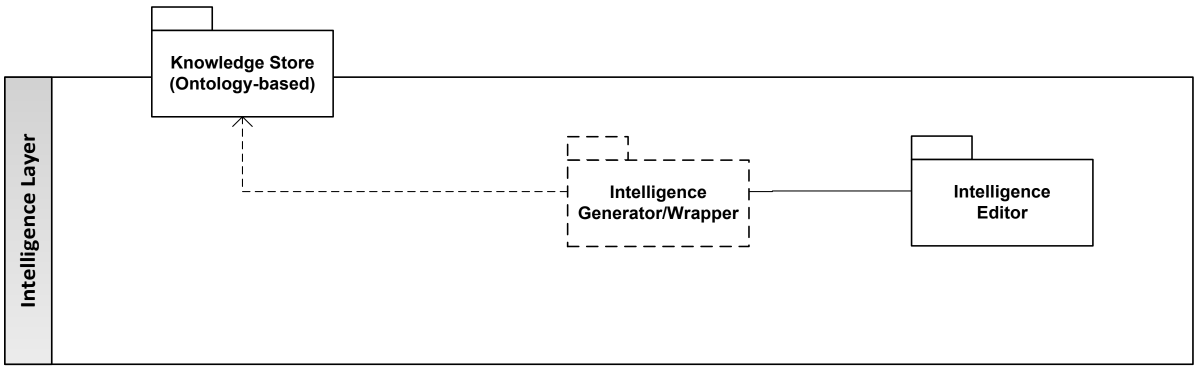

Intelligence Layer
Knowledge Store (Ontology-based)
Intelligence Generator/Wrapper
Intelligence Editor

Figure 6.8: Intelligence Layer Modules

The *Intelligence Generator/Wrapper* module, as depicted in Figure 6.8, is responsible for generating most of the necessary intelligence logic for the developed semantic services, and mapping of semantic descriptions and syntactic descriptions to the intelligent properties. The module also relies on the re-usable *intelligence logic* (i.e. agents' behaviour and operations) available through the Intelligence Editor. The operations that are essential to realize *intelligence* for semantic services are those that implement *autonomous, proactive, reactive,* and *collaborative* behaviours. The *machine-processable* property according to the definition of our intelligence semantic service, as presented in Chapter 5 (Section 5.2), is addressed by the knowledge store (cf. Figure 6.8), shared with the semantics layer for simpler interoperability amongst the agents and semantic services, and to achieve common interpretation of terminologies used in semantic descriptions and domain ontologies (Ringelstein, Franz & Staab, 2007).

The *Intelligence Editor* can be implemented using any agent-based environment that provides an environment for editing autonomous, proactive, reactive, and collaborative capabilities as generated by the *Intelligence Generator/Wrapper*. Additional details on how the intelligence layer was implemented using technologies of choice are presented in Chapter 7 (Section 7.3).

The end results of the intelligence layer are functional intelligent semantic services. These developed intelligent semantic services could then be validated and deployed into some internal or external execution environment, where it would be possible for consumers and machines to automatically discover, select, compose, invoke, execute, and manage these services.

In this study, the validation and deployment of intelligent semantic services is partially dealt with, since the focus of the project is primarily on simplifying and accelerating the process of engineering *IsS*.

In the following section, we shall describe the core components of the *i*SemServ framework in detail; these are: *service descriptions, semantic descriptions, and service intelligence*.

6.5. FUNDAMENTAL COMPONENTS

The core components that form part of our main contributions in this study focus on methods and tools that facilitate: (1) multiple language support for syntactic descriptions; (2) independent semantic descriptions generation from syntactic descriptions and service models; and (3) intelligence wrapping of semantic services to produce intelligent semantic services.

In the following subsections, we discuss the theoretical aspect of these fundamental components.

6.5.1. Syntactic Descriptions

In the absence of syntactic descriptions, a service is not available for consumption outside the service provider's environment; and it might not even be discoverable in the public domain. Without syntactic descriptions, users are unable to make prior decisions on whether or not to invoke and consume a particular service.

The ability of syntactic descriptions lies in the fact that, when available, users are able to know what inputs, outputs, pre/post conditions, and results, a particular service satisfies (Ringelstein, Franz & Staab, 2007).

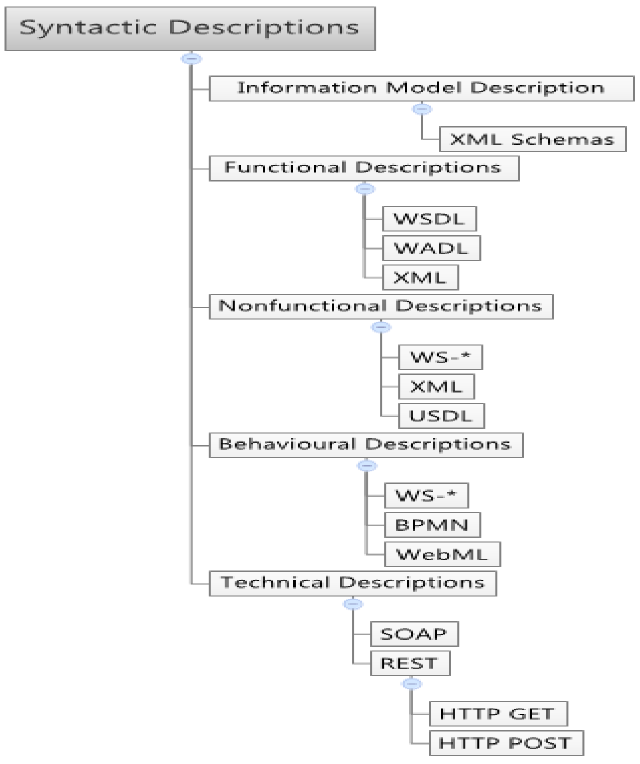


Figure 6.9: Syntactic Descriptions Contract

In the *i*SemServ framework, syntactic descriptions are defined by using the common service description contract, as defined by Vitvar *et al.,* (2009). The contract is partly illustrated in Figure 6.9. The contract indicates that it is possible to describe syntactic services using different description languages and models (e.g. WADL or WSDL).

Therefore, syntactic descriptions defined in any XML-base language need at least to conform to the descriptions contract. This means that every service needs to be described according to the *information model description*, *functional descriptions*, *non-functional descriptions*, and *technical descriptions* (Vitvar, Kopecky & Fensel, 2009). The *i*SemServ framework follows the XML Schema, as the *information description model* for defining inputs, output, error messages, and other relevant data used within syntactic descriptions. XML Schema provides an open, structured, and platform-independent approach in describing documents. This means that service description languages supported in our framework are XML-based.

In order to describe service capabilities (i.e. *functional descriptions*), WSDL(Web Service Description Language) for SOAP-based services, and WADL (Web Application Description Language) for REST-based services (Hadley, 2009) are adopted for the *i*SemServ framework. Both WADL and WSDL are XML-based, and are commonly used to describe resource-based and action-based services. *Non-functional descriptions* are additional data elements that augment service descriptions, but do not affect service functionalities (Vitvar, Kopecky & Fensel, 2009). However, they could affect the decision by the service requester as to whether to use the service or not.

These *non-functional descriptions* could range from simple data, such as the “price” of a service to complex and contextual information, such as service performance and the geographical relevance of the service. In our work, non-functional descriptions are embedded within the non-semantic descriptions, using specialized XML tags, such as `<documentation>` in the case of WSDL standard.

Technical descriptions define the communication protocols used and the messages exchanged, during service invocations (Vitvar, Kopecky & Fensel, 2009). This aspect in our work is covered through the implicit SOAP bindings in WSDL, and the unified interfaces in WADL for RESTful services, using HTTP methods, such as GET and

POST. However, *behavioural descriptions,* as defined by Vitvar *et al.* (2009), are not used in our work for syntactic descriptions. This is because behavioural descriptions represented using WS-*[36] specifications are not machine-interpretable (Ringelstein, Franz & Staab, 2007); and thus, they do not add any value to the formation of an intelligent semantic service.

As discussed in Chapter 2 (Section 2.2), syntactic descriptions are not practicable for machines or software agents, as they only state what a service does, but lack semantic information on how the service can achieve its functionalities. In the following section, we shall highlight on the contract of semantic descriptions.

6.5.2. Semantic Descriptions

The *i*SemServ framework accomplishes the generation of semantic descriptions by following a common semantic-level description contract of Vitvar *et al.,* (2009).The approach is illustrated in Figure 6.10, and is similar to the syntactic description contract. The common semantic descriptions contract prescribes that a semantic service needs to be described according to the *functional descriptions, non-functional descriptions*, and *behavioural descriptions,* using *information model descriptions.* The information model descriptions are provided in the form appropriate of *domain ontologies*, which provide common terminologies that are used across functional, non-functional, and behavioural descriptions (de Bruijn *et al.*, 2008:31).

Technical descriptions are generally not represented at a semantic level; since according to Vitvar *et al.* (2009), these are covered adequately in the syntactic descriptions. However, OWL-S represent technical descriptions through the use of service groundings (Martin *et al.*, 2004). Within the *i*SemServ framework, semantic technical descriptions are only considered when the semantic description language of choice is OWL-S.

[36] WS-* are web services specifications such as WS-BPEL, which is mainly used for syntactic workflow definitions

In this study, the following status quo holds for generating semantic descriptions: (1) *Information model descriptions* are realized through the ontology-based *knowledge store.*

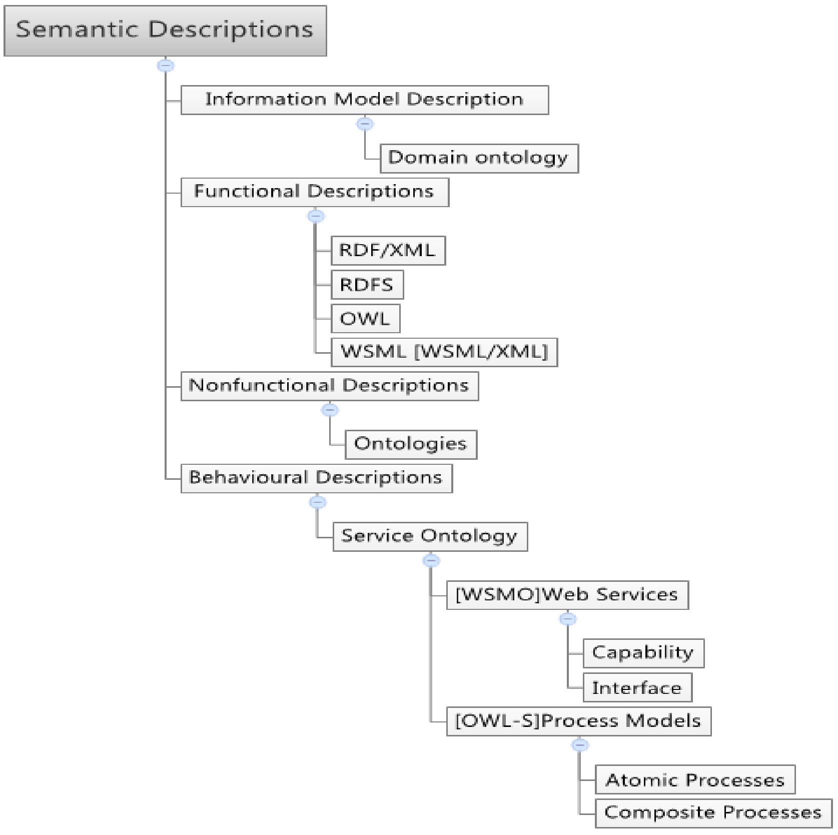


Figure 6.10: Semantic Service Descriptions Contract

(2) Although the premise of our framework is that it should support various semantic descriptions languages, including lightweight description languages, such as RDF(S). For experimentation purposes in this study*, functional descriptions* are generated, either in WSML or OWL – due to the high-level nature of their expressivity. The process of how this is achieved is detailed in Chapter 7. (3) *Non-functional descriptions* are represented either through the use of the `nonFunctionalProperty` in WSMO, or through the use of `textDescription` in OWL-S service profile class; and (4) *Semantic behavioural descriptions* detail how the interactions, through the use of input, output, conditions and message exchanges happen between the service provider and requester (de Bruijn *et al.*, 2008:32).

In the context of this study, semantic descriptions are represented through the use of formal ontologies. In OWL-S, behavioural descriptions are represented within the process model class; whilst in WSMO, this is achieved through the use of the `Interface` property within the WSMO Web Services element (de Bruijn *et al.*, 2008:33).

6.5.3. Service Intelligence

Service Intelligence is core to the proposed framework. As detailed in Chapter 5, the intelligence building block wraps semantic services with the properties of pro-activeness, reactivity, collaboration, and autonomy. This is achieved at two levels, namely: the *message-level* and the *knowledge-level*.

At the *message-level*, services and intelligent agents are mapped through syntactic descriptions. Thus, syntactic descriptions, particularly *functional and technical descriptions,* are used to share information, such as input, output, messages, and communication protocols with software agents for the purposes of automating and enhancing service consumption. Since WSDL uses protocols (e.g. SOAP) that are incompatible with software agents languages, such as FIPA-ACL (Protogeros, 2008), messages and protocol translations could also be done at this level, such as *SOAP2ACL* and *ACL2SOAP* (Hemayati *et al.*, 2010).

However, for REST-based services, the translation of protocols does not apply, because open and common HTTP methods are used, such as GET and POST (Pautasso, Zimmermann & Leymann, 2008). These do not need any translation at message-level, when agents are communicating with syntactic services.

Figure 6.11 demonstrates the high-level components for intelligence wrapping. These include the *structure,* which is mainly the non-functional descriptions of an agent, including details, such as the *agent name* (Zhu & Shan, 2005). *Operations* represent the behaviours that implement intelligence according to the *IsS* definition. *Messages* capture requests (e.g. goals) and responses during message-level interactions. The *knowledge base* component contains internal knowledge of an agent, which is interoperable with the common knowledge-based store.

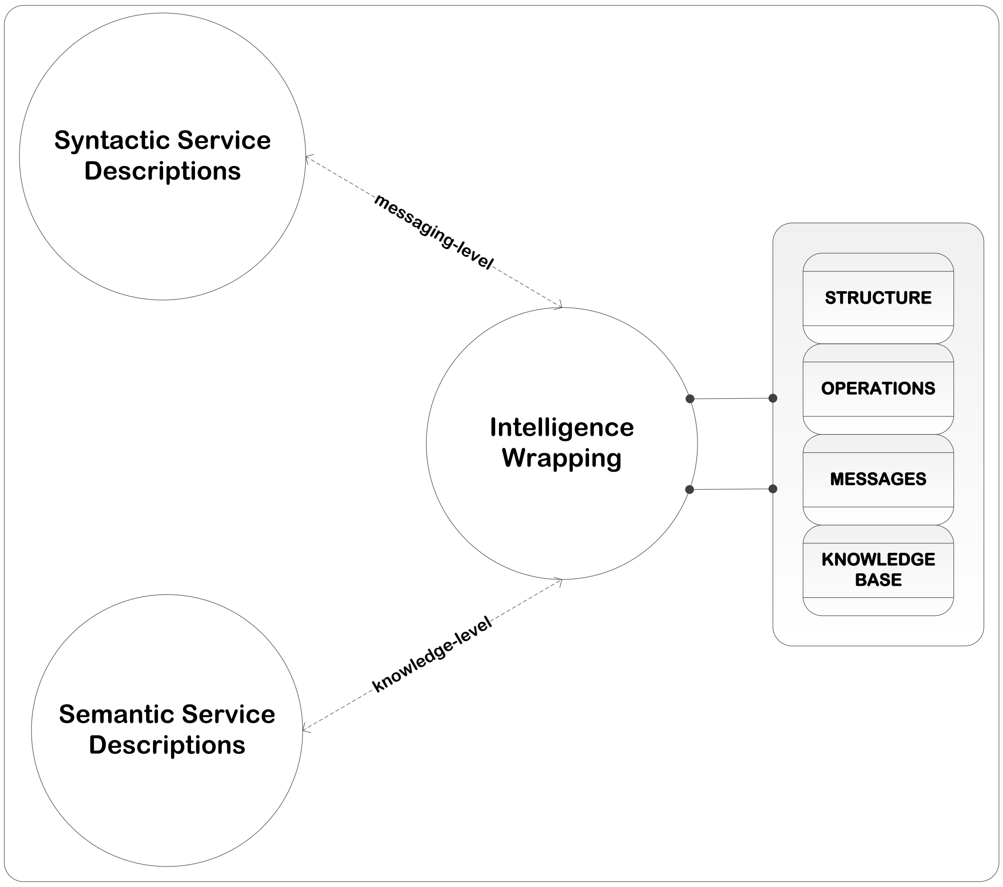


Figure 6.11: Intelligence Wrapping Overview

It is important to understand that *intelligence wrapping* in our context is considered mainly for enabling interoperation between semantic services and agents; thereby, enabling protocol translation from one format to another, and ultimately enabling agents to communicate with semantic services, both at message-level and at knowledge-level.

Intelligence wrapping does not replace the semantic service, but augments the service with behaviours that make it possible for consumers (e.g. humans and software machines) to automatically perform a number of activities with regard to semantic services. The addition of intelligence to semantic services also happens in a decoupled manner; that is, semantic services are not tightly linked to agents, and could still be invoked and executed independently.

6.6. SUMMARY

In this chapter, we presented and described an *i*SemServ framework that simplifies and accelerates the process of engineering intelligent semantic services. The design principles of the framework may be grouped as follow; *simplification, acceleration*, and *intelligence*. The framework was also designed with a model-driven engineering methodology in mind, which promotes decoupling, interoperability, seamless translations between syntactic and semantic descriptions, and code stub generation at various layers.

The core contributions of the framework are: (1) seamless development of syntactic service descriptions, based on an XML-based common descriptions contract; (2) Semantic descriptions generation in multiple semantic models (e.g. WSML and OWL) following a common semantic description contract; and (3) the wrapping of semantic services with intelligent properties to achieve machine-interpretable intelligent semantic services.

The next chapter will cover, the implementation of the *i*SemServ framework, including all its components, processes, and technologies.

CHAPTER 7: iSemServ Framework Implementation

The implementation specifics of the proposed service creation framework are illustrated and described in this chapter. This is preceded by the illustration and description of the *i*SemServ technical architecture, and the main technologies that are essential to the overall implementation of the different layers as discussed in Chapter 6. The implementation in this chapter focuses on the development of the UML profile for model annotations, design and development of the code generation rules and templates, and the intelligence wrapping logic – all responsible for realizing functional intelligent semantic services.

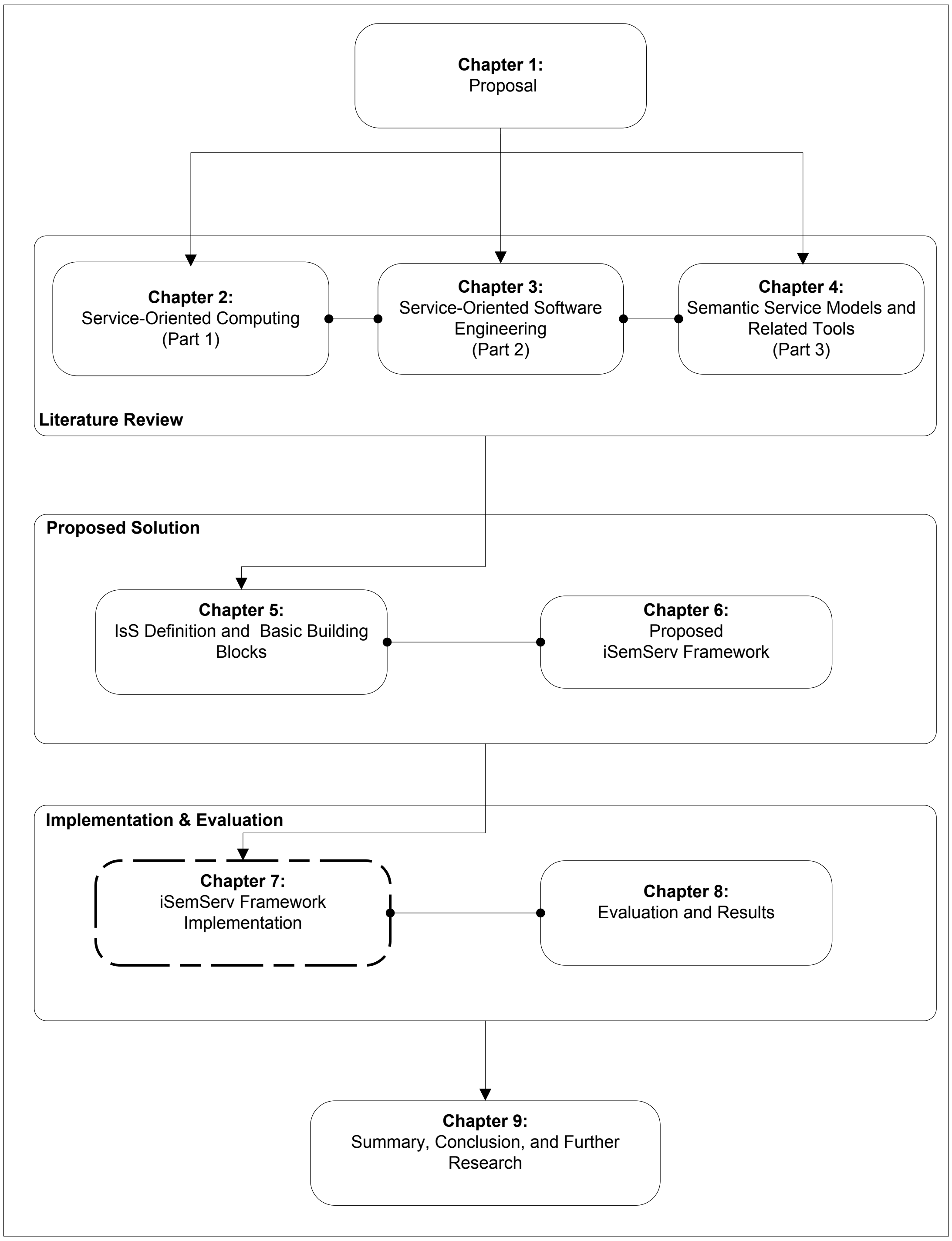

Chapter 1:
Proposal
Chapter 2:
Service-Oriented Computing
(Part 1)
Chapter 3:
Service-Oriented Software Engineering
(Part 2)
Chapter 4:
Semantic Service Models and Related Tools
(Part 3)
Literature Review
Proposed Solution
Chapter 5:
IsS Definition and Basic Building Blocks
Chapter 6:
Proposed iSemServ Framework
Implementation & Evaluation
Chapter 7:
iSemServ Framework Implementation
Chapter 8:
Evaluation and Results
Chapter 9:
Summary, Conclusion, and Further Research

Figure 7.1: Overall Thesis Structure

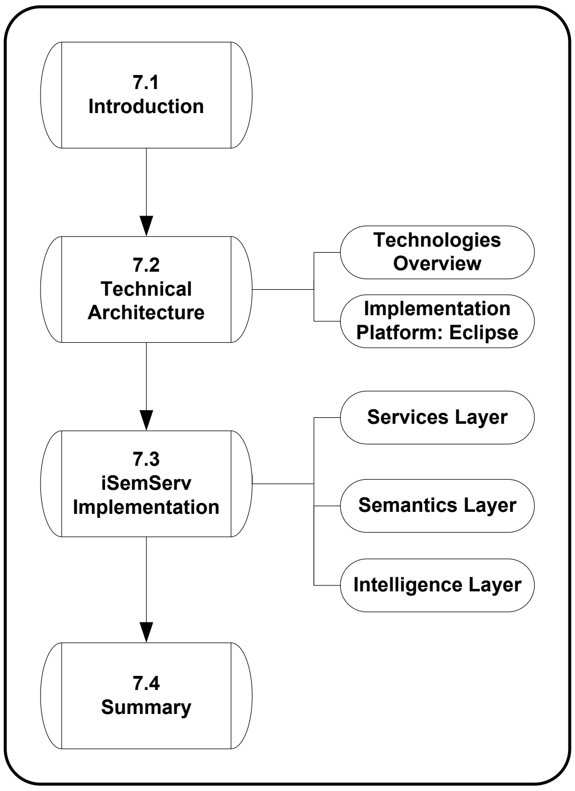


Figure 7.2: Chapter 7 Layout

7.1. INTRODUCTION

This chapter discusses the proof-of-concept implementation and technologies exploited to implement the components of the proposed *i*SemServ framework. In Chapter 8 the actual use of the framework is demonstrated by means of an appropriate use case scenario.

The framework was implemented using the Eclipse platform, which encompasses a variety of re-usable service engineering components. Although the *i*SemServ framework is touted as platform independent and could be implemented using any other SOA-based platform, the decision to implement the framework using the Eclipse[37] environment was motivated by a number of factors and benefits, such as: (1) Openness; (2) wider support and community involvement; (3) the availability of plug-ins; (4) the support of multiple programming and modelling languages, (5) simple extensibility; and (6) the wider adoption and use by service developers.

This chapter is organized as follows: Section 7.2 presents and discusses the *i*SemServ technical architecture, which represent the high-level technological view of the framework. In addition, an overview of the methods and technologies used to realize the different layers of the framework is presented. It should be noted that the technologies used for implementing the *i*SemServ framework may differ from one implementation to the other depending on various choices and requirements. Thus, in Section 7.2, only an overview of these technologies is presented, without focusing too much on the technical details of the specific technologies. In Section 7.3, a proof-of-concept implementation is provided, according to each layer of the *i*SemServ framework. In this section, the focus is on the design and development of the UML profile for model annotations, code generation rules and templates, and the intelligence wrapping logic.

Section 7.4 summarizes the chapter by highlighting some of the lessons learnt during implementation, and some possible future improvements to the proposed framework.

[37] See: http://www.eclipse.org

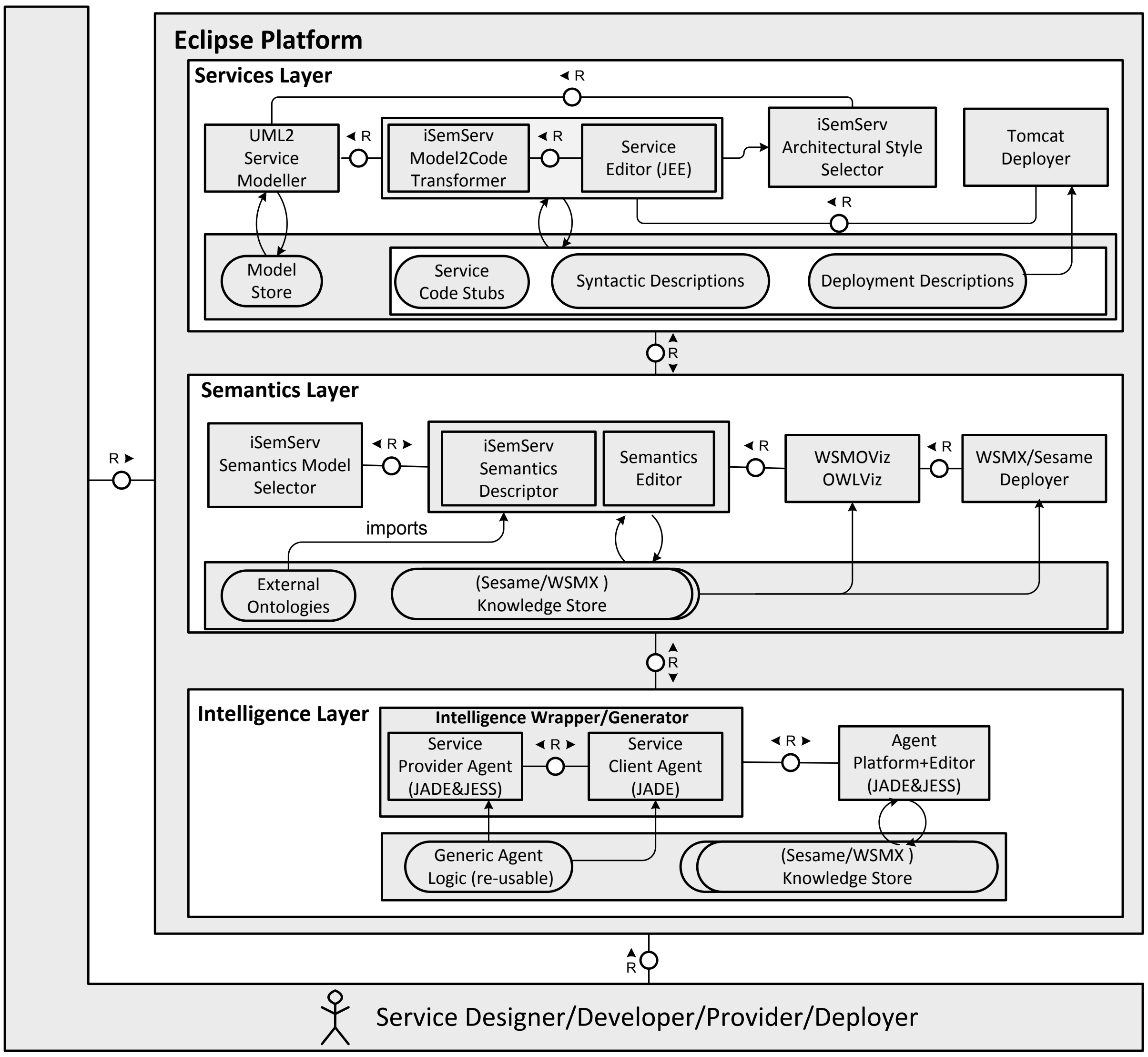
7.2. iSEMSERV TECHNICAL ARCHITECTURE

In this section, the *i*SemServ technical architecture is described. A conceptual view of the proposed framework in terms of the technological components that are relevant to the implementation is also provided.

The technical architecture in Figure 7.3 depicts a unified technology infrastructure, meant to ease and accelerate the process of engineering intelligent semantic services. The architecture is similar to the conceptual service creation framework. It is also made up of three integrated, but independent, layers. The technical specifications of each layer are discussed in Section 7.3.

7.2.1. Technologies Overview

In this section, an overview of all the salient technologies used to realize the *i*SemServ service-creation framework is presented. Details on how some of the technologies were applied are made available in Section 7.3 under each specific layer.



Eclipse Platform
Services Layer
UML2 Service Modeller
iSemServ Model2Code Transformer
Service Editor (JEE)
iSemServ Architectural Style Selector
Tomcat Deployer
Model Store
Service Code Stubs
Syntactic Descriptions
Deployment Descriptions
Semantics Layer
iSemServ Semantics Model Selector
iSemServ Semantics Descriptor
Semantics Editor
WSMOViz OWLViz
WSMX/Sesame Deployer
imports
External Ontologies
(Sesame/WSMX) Knowledge Store
Intelligence Layer
Intelligence Wrapper/Generator
Service Provider Agent (JADE&JESS)
Service Client Agent (JADE)
Agent Platform+Editor (JADE&JESS)
Generic Agent Logic (re-usable)
(Sesame/WSMX) Knowledge Store
R
Service Designer/Developer/Provider/Deployer

Figure 7.3: *i*SemServ Technical Architecture

7.2.1.1. Services layer

In this sub-section, we briefly highlight the technologies that were used to implement the services layer.

- ***Service Modelling:*** The Unified Modelling Language (UML2) Eclipse plug-in was used to enable the modelling of services. The plug-in allows for the creation of all UML diagrams (e.g. class, activity, and use cases). In the context of the proposed service creation framework, the UML2 plug-in facilitates the model-driven, complexity hiding, and visualization design principles.

- ***iSemServ Model2Code Transformer:*** This is an Acceleo[38] compliant code generation module implemented particularly for the proposed solution. In brief, Acceleo (Acceleo, 2011) is an open source model-to-text language (MTL) framework. It provides a flexible and simple environment for designing and developing a variety of code generators, using simple and standard templates. It is used in the proposed solution, primarily because its design principles are based on increasing software development productivity.

- ***JAX-RS:*** a Java API for RESTful (Representational State Transfer) Web Services specification (Sun Microsystems, 2009b) is adopted as a standard for realizing RESTful services within the services layer. Jersey[39], as reference implementation for JAX-RS, is chosen for handling the development and deployment of RESTful services.

- ***JAX-WS:*** a Java API for XML-Based Web Services specification (Sun Microsystems, 2009a) is used as a standard for implementing SOAP-based services in the services layer. The reference implementation selected for the realization of JAX-WS is Metro[40], which is compatible with the Apache Tomcat deployment environment. Tomcat is used as the appropriate service execution environment in our implementation for deploying JAX-RS and JAX-WS services.
- ***iSemServ Architectural Style Selector:*** This is a module that automatically detects the type of modelled services (e.g. RESTful or SOAP). The detected service type is used by the *Model2Code Transformer* and the *Syntactic Descriptor* modules to generate the relevant service code stubs (e.g. JAX-RS or JAX-WS), syntactic descriptions (e.g. WADL or WSDL), and deployment descriptions (e.g. XML). Deployment descriptions are essential for deployment purposes, such as to the Tomcat environment.

- ***Java Enterprise Edition (Java EE) editors:*** Eclipse embedded Java editors are used to provide the developer with an environment in which to complete the behaviour/logic of the service, as generated from UML service models.

[38] For more details visit: http://eclipse.org/acceleo/
[39] JAX-RS (Jersey) reference implementation available at: https://jersey.dev.java.net/
[40] JAX-WS (Metro) reference implementation available at: http://metro.java.net/guide/

7.2.1.2. *Semantics layer*

In this sub-section, we briefly highlight the technologies that were used to implement the semantics layer.

- ***iSemServ Semantics Model Selector:*** This is a simple UML profile-based module, specifically implemented for the proposed service creation framework. It allows the service developer to choose from a variety of semantics models (e.g. WSMO or OWL-S) when designing services.

- ***iSemServ Semantics Descriptor:*** This is another component developed for the *i*SemServ framework for the purpose of automatically generating skeleton[41] semantic descriptions, based on the preferred semantic model (e.g. WSMO) as inferred by the *Model Selector*.

- ***Semantics Editor***: A number of external semantics editors for different semantic models are used for editing auto-generated semantics and domain knowledge. These include WSML editor and OWL-S editor, which provide developers with useful features, such as error detection, syntax auto-completion, and highlighting, when editing generated service semantics.

- ***WSMOViz/OWLViz:*** Keeping in line with the visualization and complexity hiding principles, existing ontology visualization approaches are used to aid the developer in understanding, viewing, and editing complex semantic descriptions. For WSMO-based semantics, WSMOViz (Kerrigan, 2006), is embedded within the Eclipse IDE. OWLViz[42] is used to support the developer in visualizing OWL-based ontologies.

[41]By *skeleton semantics* we mean partial semantic descriptions and/or domain ontologies that could be augmented by the developer.
[42]http://www.co-ode.org/downloads/owlviz/OWLVizGuide.pdf

7.2.1.3. Intelligence layer

In this sub-section, we briefly highlight the technologies that were used to implement the intelligence layer.

- ***JADE:*** This is a Java-based agent development environment and middleware (Ye & Yang, 2009). The environment is integrated within Eclipse to realize the *autonomous*, *machine-processable*, and *collaborative* capabilities necessary for producing intelligent semantic services. In brief, JADE[43] is made up of a platform, consisting of containers, responsible for managing and executing agents. JADE also provides libraries for implementing agent-based logic necessary for realizing partial intelligence (e.g. autonomy and collaborations). It was chosen for *i*SemServ framework implementation mainly because of its open nature and use of Java programming language, which is fully supported by Eclipse.

- ***JESS:*** It is referred to as a Java Expert System Shell (Balachandran, 2008). It is generally used as a rule-based engine and programming environment using Java (Friedman-Hill, 2003). It is touted as a "powerful tool for systems with intelligent reasoning abilities" (Balachandran, 2008), using algorithms, such as Rete[44]. In this study, JESS is employed in conjunction with JADE for the purposes of realizing the *proactive* and *reactive* properties of an intelligent semantic service. In addition, it is exploited to enable the developer to define free-format reasoning rules that enable proactive and reactive capabilities within a specific intelligent semantic service. These rules can be expressed in JESS rule language or XML[45], and are stored in a text file with an extension ".clp". The language is expressive and capable of defining logical relationships, using minimal code. Moreover, the language has built-in functions that can easily be reused. JESS also has functions that enable direct access to Java APIs. Such reasons make JESS one of the suitable alternatives for defining rules that could be used by software agents to reason over domain and service ontologies.

[43] More information on JADE: http://jade.tilab.com/

[44] See: http://www.jessrules.com

[45] See: http://www.jessrules.com

Furthermore, in this study, JESS was chosen because of its easier integration with Java, JADE, and Eclipse.

7.2.2. Implementation Platform: Eclipse

The *i*SemServ framework was implemented as a collection of unified plug-ins using the Eclipse platform. Eclipse is a mature, well-designed, and extensible platform. This platform supports developers with libraries to build components that interoperate seamlessly. The key to the seamless integration of tools in Eclipse is the plug-in approach(Rivières & Wiegand, 2004). The plug-in modules developed for the *i*SemServ framework integrate with Eclipse in exactly the same way as any other Eclipse plug-in, without much effort from the developer.

The following section discusses the *i*SemServ framework implementation particulars.

7.3. *i*SEMSERV IMPLEMENTATION

The *i*SemServ framework was implemented, using a variety of open source (OS) service-based modules within the Eclipse environment, as indicated in the preceding sections. The OS service-based modules include existing and newly developed modules, as highlighted in Section 7.2.

Adhering to the decoupling and separation of concerns principles, the implementation was realized in phases. Thus, each layer was implemented independently of any other layers, and could also be used independently of other layers. Nevertheless, the completed implementation involved the integration of all the layers into one operational *i*SemServ Eclipse plug-in.

In terms of the implementation strategy, the services layer was implemented first, as it handles the initial processes of creating syntactic services. The semantics layer, which is responsible for integrating service ontologies into services engineered within the services layer, then followed. The intelligence layer, which consists of components necessary to incorporate intelligence into the semantic service, was implemented last. Although the implementation was meant to demonstrate merely a proof-of-concept, and not a fully-fledged *i*SemServ framework, an effort was made to implement many of the salient features deemed necessary for simplifying and accelerating the process of building up intelligent semantic services.

7.3.1. Services Layer

The services layer supports the service engineer with the necessary modules for producing standard-independent syntactic services. As illustrated in Figure 7.3, different modules are necessary for the complete functioning of the services layer. In this section, the implementation details of these modules are discussed.

7.3.1.1. Service Modeller

The *service modeller*, which represents the core module responsible for capturing the internal and external properties of the identified service(s), was implemented using the UML2 development kit integrated within the Eclipse platform. In this module, the service designer could capture the structure of services using UML class diagrams. The behaviour of the services could also be captured using UML activity diagrams at this layer.

It should be noted as well that within the *i*SemServ platform, the designer could also import external service models (i.e. UML class diagrams) from the model store, or as designed, using any other UML2-compliant tool, such as ArgoUML[46]. Nevertheless, for the purposes of the service creation framework implementation, the premise is that new services should be designed using the UML2 *service modeller* module.

In general, service models can be designed using any modelling language of choice. However, Model Driven Architecture (MDA) compliant languages, such as the Unified Modelling Language (UML), are encouraged by the Object Management Group (OMG) (OMG, 2010b). Thus, for the implementation of the service modeller module, UML-compliant models were chosen. This is mainly because of their wide spread use in industry and academia, and their support for platform independency, ensuring "*portability, interoperability, extensibility and re-usability through an architectural separation of concerns between the specification and implementation"* (Lautenbacher, 2006).

[46] See: http://argouml.tigris.org/

For the proper functioning of the *service modeller* module, the class diagrams capturing the properties of services to be engineered need to be modelled based on the *i*SemServ UML profile. The specific *i*SemServ UML profile implemented for the *service modeller* module is depicted in Figure 7.4.

In brief, UML profiles are a group of custom keywords (i.e. stereotypes), data types, constraints, and tagged values that could be used to annotate and extend UML diagrams (Bensaber & Malki, 2008). Moreover, the distinct stereotypes within the UML profile provide the necessary flexibility to annotate the model in a manner that would promote different representations of the model.

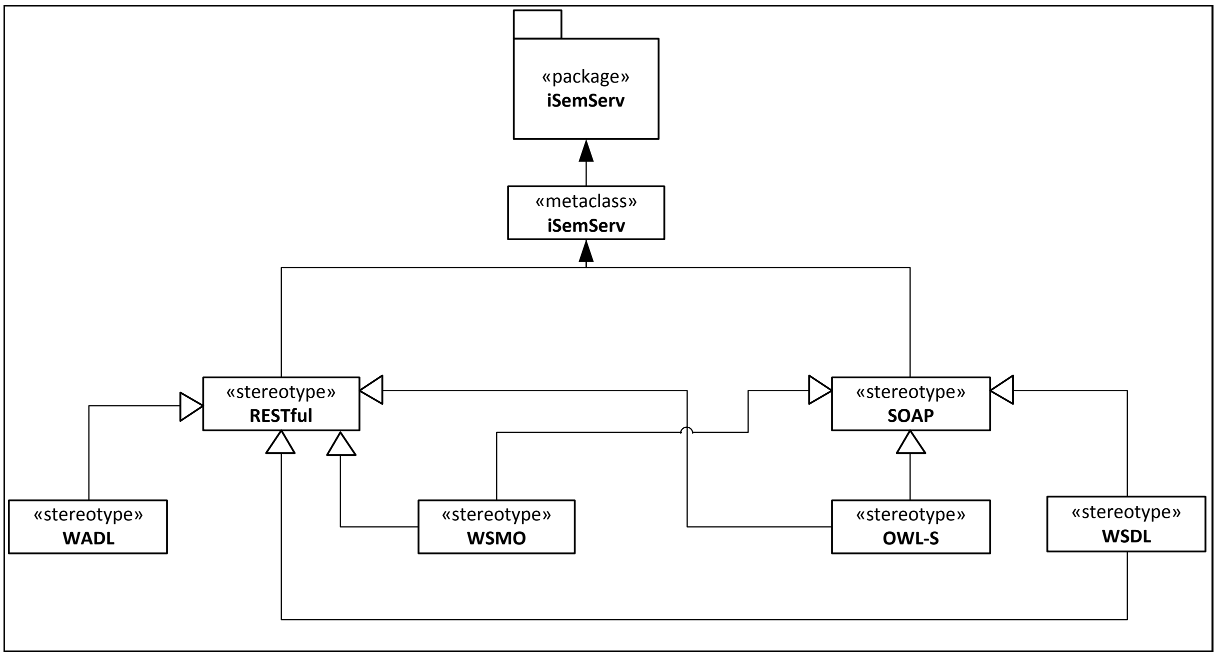


Figure 7.4: Partial *i*SemServ UML Profile

In Figure 7.4, a UML profile implemented for the proposed *i*SemServ framework is illustrated using a simplified UML package diagram. This is essential for Java code generations, where classes are generally organized within packages. The key stereotypes in the *i*SemServ profile are **<<RESTful>>** and **<<SOAP>>**. These two stereotypes can only be applied to class diagrams. This means that any class diagram capturing the structure of a service could be annotated as **<<RESTful>>**or **<<SOAP>>**, the two common Web services standards to date. The other stereotypes deriving from the main stereotypes are **<<WADL>>** and **<<WSDL>>**, which enable the developer to decide on the syntactic descriptions to be generated for the modelled service(s). The use of the profile is demonstrated in Chapter 8 (Section 8.2).

As illustrated in Figure 7.4, both **<<WSMO>>** and **<<OWL-S>>**stereotypes are catered for within the *i*SemServ framework implementation for selecting the skeleton semantic descriptions and domain ontologies to be auto-generated. Additional keywords can also be added to accommodate other syntactic descriptions (e.g. USDL (Cardoso *et al.*, 2010)) and semantic descriptions or annotation standards, such as WSDL-S (Akkiraju *et al.*, 2005).

For the implementation of the *i*SemServ framework – and adhering to the principle of supporting multiple languages, UML profiles are also viewed as significant. For instance, within one service model, a service designer could annotate the model to represent RESTful services and WSMO semantic descriptions. Similarly, the same model could be annotated to represent SOAP services and OWL-S semantic descriptions. It is also important to note that UML profiles are easily implementable, using any UML compliant tool, and could be extended by adding new keywords and other details of interest.

7.3.1.2. iSemServ Model2Code Transformer

Once a service model is defined according to the *i*SemServ profile, it is inputted into the implemented *iSemServ Model2Code Transformer*. This module is capable of automatically determining the type of a service, syntactic descriptions, and deployment descriptions to be generated - given a UML2 compatible model. The generator relies on the steps depicted in Figure 7.5.

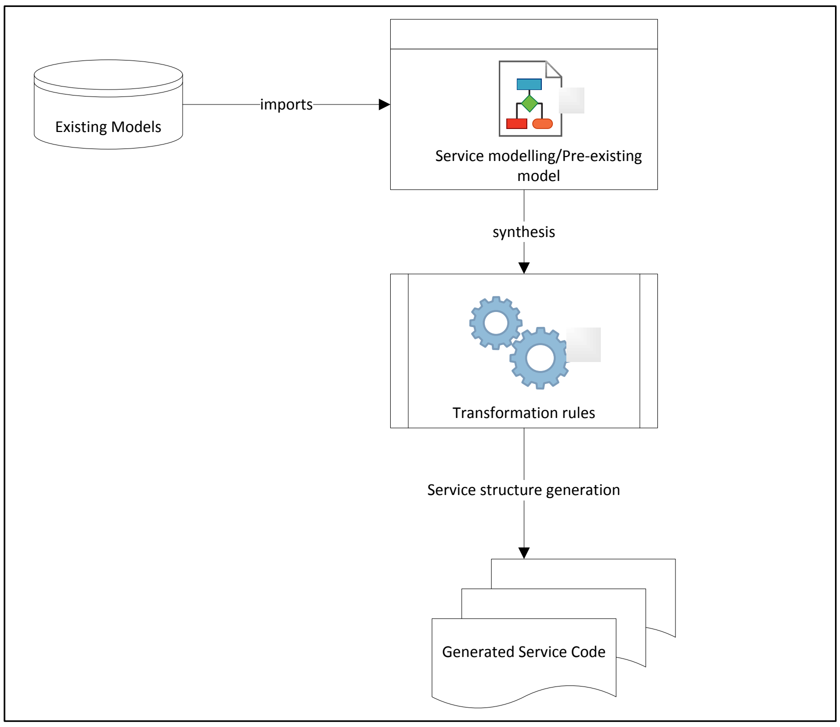


Figure 7.5: Service Code Generation Steps

The *Model2Code Transformer* was implemented using a number of code transformation rules and static code templates solely defined for the *i*SemServ framework. The code generation templates implemented for the proposed framework are based on the Acceleo platform integrated within the Eclipse IDE, as discussed in Section 7.2.1. The templates were implemented using the model-to-text scripting language and Java custom services/methods such as `hasStereotype(String Keyword)`.

The transformation rules that are the foundation of the *Model2Code Transformer* within the services layer include:

- **model2services**: These are the mappings that are meant to transform appropriately annotated service models to the necessary service code structure. In our context, this would either be JAX-RS or JAX-WS. For JAX-RS, that is, a service model annotated with the **<<RESTful>>** stereotype, the mappings are tabulated in Table 7.1.

Table 7.1: Model2Service Mappings (RESTful)

RESTful Methods	UML Operations Start Keywords
@GET	get, find, request, search, check
@PUT	put, update, set, place, edit, modify
@POST	post, create, add, do
@DELETE	delete, cancel, remove

For example, these mappings can be translated as follows: For any UML operation that starts with the keyword "**get**", an associated **@GET** method would be annotated to the relevant JAX-RS method. Any other UML operation that does not start with any of the listed keywords (cf. Table 7.1) in the case of **<<RESTful>>** annotated UML classes would be transformed into a normal POJO (Plain Old Java Object) method. The remaining transformation mappings for **model2services** are adopted from the JAX-RS and JAX-WS specifications (refer to: (Sun Microsystems, 2009a; Sun Microsystems, 2009b)), as partially depicted in Table 7.2.

Table 7.2:UML2JAX-RS Mappings

UML Class Diagram	Java Classes (JAX-RS)
<<RESTful>>class	RESTful Service (@Path)
[keyword] operation	@[GET|PUT|DELETE|UPDATE]
operation parameters (in)	@Consumes
operation parameter (return)	@Produces
Operation parameters (in)	@[Path|Query|Form|Header|Cookie]**Param**

Table 7.3 shows the mappings that are applied in the proposed framework to transform UML service models annotated with a **<<SOAP>>** stereotype to SOAP-based services, using JAX-WS specifications. For any UML class that is annotated with the **<<SOAP>>** stereotype, an equivalent Java-based SOAP service, annotated with the **@WebService** keyword, is generated. In addition, for all the operations of the **<<SOAP>>** annotated UML class, equivalent Java methods would be annotated with the **@WebMethod** as prescribed by JAX-WS.

Table 7.3: UML2JAX-WS Mappings

UML Class Diagram	Java Class (JAX-WS)
<<SOAP>> class	SOAP Service (@WebService)
Class Name	Service Name
Package Name (reversed)	TargetNameSpace
Owned Operation	@WebMethod
Package Name +Interface	EndpointInterface

- **model2descriptions:** These mappings enable the *Model2Code Transformer* to automatically derive complete syntactic descriptions from the UML class diagrams. For example, if the service model has been annotated with the **<<WADL>>** stereotype, the mappings in Table 7.4 would apply when the Model2Code transformer is automatically generating syntactic descriptions.

Table 7.4: UML2WADL Mappings

UML Class Diagram	WADL Descriptions
<<RESTful>><<WADL>> class	<application><resources>
operation	<resource><method>
operation parameters (in)	<request><param …/><representation>
operation parameter (return)	<response><representation>
representation tag value	<representation>

These mappings comply with the specifications that prescribe the approach for describing RESTful services using WADL. For instance, for any UML class that is annotated with both <<RESTful>> and <<WADL>> keywords, an equivalent WADL description file would be generated using the mappings as shown in Table 7.3. The practical usage of these mappings is demonstrated in Chapter 8 (Section 8.2)

- **model2deployment:** These are the simple mappings that allow the *Model2Code Transformer* (cf. Figure 7.3) to create the relevant deployment description files from the service models. For implementation purposes, only the deployment descriptions targeting (e.g. JAX-RS or JAX-WS) the Apache Tomcat application server can be generated using the *i*SemServ framework. These mappings are similar to those tabulated in Table 7.4, with the main difference being in the names of the XML tags.

In order to augment or edit the generated service code, descriptions, or even deployment description files, the service engineer is provided with existing editors, such as Java EE and XML Editors integrated within the Eclipse IDE. This is in accordance with the re-usability and interoperability *i*SemServ framework design principles.

The practical demonstrations of the various modules' implementation of each layer are illustrated in the next chapter, using different use case scenarios. In the following section, the details related to the implementation of the semantics layer are presented. (In order to review the high-level view of the modules within the semantics layer, please refer back to Chapter 6 or see Figure 7.3.)

7.3.2. Semantics Layer

Semantic services are realized through the use of domain ontologies and semantic descriptions. Thus, once syntactic descriptions are delivered in the services layer, the automatic generation of partial semantic descriptions is the focus within the semantics layer. The essential components that were implemented for the proposed framework are described in the following sub-sections.

7.3.2.1. iSemServ Semantics Model Selector

The proposed framework needs to subscribe to the principle of semantic standards independency. This is to ensure that as semantic standards develop, the proposed framework remains relevant. As noted in the previous sections, there have been a number of efforts with regard to models that provide methods and tools for building domain ontologies and semantic descriptions.

Because of the diversity of semantic models, an *i*SemServ *Semantics Model Selector* was implemented, mainly using the *i*SemServ UML Profile. The *Model Selector* mainly infers the semantic model of choice, based on the service model's annotations. This simply means that the developer annotates the service model, using specific stereotypes, such as **<<OWL-S>>** within the services layer, to explicitly indicate what type of skeleton semantic descriptions and/or domain ontologies need to be automatically generated.

The *Model Selector* does not restrict the number of semantic models that could be selected simultaneously. Thus, multiple semantic models could be selected for one specific service model.

It should also be noted that in this context, the service engineer is only expected to make the semantic models preference visible through the service model, as defined in the services' layer. If the service model is not annotated with specific semantic models' stereotypes, domain ontologies and/or semantic descriptions can be imported from existing sources. This is essential for complying with the re-usability principle, since the generation of domain ontologies is also quite complex and resource intensive, and normally involves a number of domain experts (Sabou *et al.*, 2005). Thus, building new domain ontologies for every new service is, in many instances, discouraged, but re-using well-defined and generic pre-existing domain ontologies is encouraged.

7.3.2.2. iSemServ Semantics Descriptor

The *Semantics Descriptor* uses the details gathered by the *Semantic Model Selector* to automatically generate the skeleton semantic descriptions, and in preferred cases, domain ontologies for the planned services. This module was implemented, based on a number of transformation rules, using the Acceleo framework. The main rules include:

- **model2ontologies:** Domain ontologies are the cornerstone of semantic services. Although the intentions in this study are not to build complete domain ontologies; for the *i*SemServ framework, an implementation was conducted to support the process of automatically generating partial domain ontologies from the service model(s). This was achieved by using the class properties, types, operations, and their input parameters to partially infer some of the common concepts relevant to the service(s) being engineered.

Generic mappings for WSMO and OWL-S were also defined for the *i*SemServ solution. In Table 7.5, the mappings between the UML class diagrams and the WSMO domain ontologies are listed. These are based on the salient components (i.e. ontologies) of WSMO, as discussed in Chapter 4.

Table 7.5: UML2WSMO Mappings

UML Class Diagram	WSMO Domain Ontologies
<<WSMO>><<ontology>> Stereotype	WSMO Domain Ontologies
Class name	Ontology name, Concept
Class properties, Operation parameters (in)	Concepts' attributes
Operations, Enumerations, DataTypes	Concepts
Association/Composition/Aggregation	Relation
Generalization	Sub-Concept
EnumerationLiterals, DefaultValues	Instances
Constraints	Axioms

For instance, every class name annotated with the <**<WSMO>>** and **<<ontology>>** stereotypes, specific class operations, and enumerations, are mapped to the relevant WSML ontological concepts. The properties of a class and the input parameters of operations are then mapped to the attributes of the relevant concepts. For example, in Listing 7.1, the `concept` "person" would have been generated from a class with the name "person". The attributes `firstName` and `lastName` would have been the properties of the same class. Elaborative examples are provided in Chapter 8 (Section 8.2)

```
concept person
nonFunctionalProperties
        dc#description hasValue "WSML description of a person"
endNonFunctionalProperties
   firstName ofType _string
   lastName ofType _string
```

Listing 7.1: WSMO Concept and Attributes

Furthermore, WSML relations, sub-concepts, and instances are derived from UML class relationships (e.g. aggregation, and generalization) and enumeration default values. For example, a UML child class that derives from a super class is mapped as the Sub-Concept of the specific super class. Logical expressions (i.e. axioms), which are essential when defining various WSMO elements, are mapped against the constraints defined for the specific UML class diagram.

All these mappings were implemented, using the WSMO ontology template (see: Listing 7.2), and the transformation rules realized by exploiting the Acceleo MTL (model-to-text language) and a variety of Java common services.

```
ontology = 'ontology' id? header* ontology_element*
ontology_element = concept
                | relation
                | instance
                | relationinstance
             | axiom
```

Listing 7.2: WSMO Domain Ontology Structure

For the purposes of the implementation, and for showing the applicability of supporting multiple languages, OWL-S transformation rules were also defined and implemented. If another domain ontology language needs to be supported by the suggested framework, only the relevant mappings and new stereotypes need to be defined.

- **model2semantics:** Semantic descriptions enable services to be automated, as they describe the functional, non-functional, and behavioural properties of services in using domain ontologies (Vitvar, Kopecky & Fensel, 2009). In the proposed framework, semantic descriptions could be defined using different semantic description languages similar to the process of building domain ontologies. For proof-of-concept purposes, only OWL-S and WSMO semantic descriptions were accommodated within the *i*SemServ framework.

As discussed in Chapter 4 (Section 4.2.1), OWL-S provides three elements for semantically describing Web services. The *Service Profile* semantically describes the functional aspects of the service (i.e. what does the service do?), whilst the *Service Model* semantically describes the behaviour of the service, enabling the requesting agent to understand how to interact with the service using the *Service Grounding* element.

Table 7.6: UML2OWL-S Mappings (Service Profile)

UML Class Diagram	**OWL-S Profile**
<<OWL-S>><<profile>> Stereotype	OWL-S Profile
Class name	Class, Service Name, Profile Name
Operation parameters (in) (return)	&process (#input, #output, #parameter , #results)
Generalization	SubClassOf
Constraints	&expr (#condition), restrictions, cardinality
Enumerations, EnumerationLiterals	Collections, OneOf

Table 7.6 lists some of the mappings implemented within the *Semantics Descriptor* module for transforming an **<<OWL-S>><<profile>>** annotated UML service model into OWL-S Profile descriptions. Every OWL-S annotated class name is mapped to an OWL-S class, Service Name, and Profile Name according to the OWL-S specifications (Martin *et al.*, 2004). UML class operation's input and output parameters are mapped to OWL-S Profile properties such as Inputs, Output, Parameters, and Results. UML-defined constraints are then aligned to OWL-S logical expressions in Preconditions and Effects.

Table 7.7: UML2WSMO (Semantics)

UML Class Diagram	WSMO Web Services
<<WSMO>><<webService>> Stereotype	WSMO Web service descriptions
Operation	Web service name
Operation	Capability name
Class properties, Operation parameters (in)	Shared Variables
EnumerationLiterals, DefaultValues	Preconditions, Assumptions, Postconditions, and Effects
Constraints	Preconditions, Assumptions, Postconditions, and Effects

In implementing WSMO semantic descriptions, the mappings in

Table 7.7 were formulated and realized using the WSMO template (de Bruijn *et al.*, 2005c), as presented in Listing 7.3.

```
capability = 'capability' id? header* sharedvardef? pre_post_ass_or_eff*
sharedvardef = 'sharedVariables' variablelist
pre_post_ass_or_eff =  'precondition' axiomdefinition
                       | 'postcondition' axiomdefinition
                       | 'assumption' axiomdefinition
                       | 'effect' axiomdefinition
```

Listing 7.3: WSMO Web Service Template

The mappings are mainly applicable to **<<WSMO>>** annotated UML service models in the context of the *i*SemServ framework. Each UML class operation name is directly mapped to a WSMO Web service name and one WSML Capability name. This is because each WSMO Web service (i.e. semantic descriptions) can have only one capability that semantically represents the functionality provided by the service (de Bruijn *et al.*, 2005c).

The WSMO-shared variables are mapped from UML class attributes/properties, and class operation input parameters. *Preconditions*, *post-conditions, assumptions* and *effects* are directly mapped to UML-defined constrains, class attributes' default values and enumeration literals. As mentioned, the practical demonstrations of all the transformation rules will be presented in the next chapter.

7.3.2.3. Semantics Editors

The *Semantics Editors* module was implemented by re-using the existing WSML editors and various OWL-S editors, such as Profile Model Editor and Process Model Editor (Srinivasan, Paolucci & Sycara, 2006). The main task conducted in this regard was the actual integration of these existing editors into the Eclipse platform. The purpose of the editors is to enable the service engineer to easily and uniformly review, edit, and augment the generated domain ontologies and the semantic descriptions. This is done in accordance with the reusability, interoperability, extensibility, and uniformity principles that are significant for simplifying and accelerating the process of engineering intelligent semantic services. In addition, *Semantics Editors* are also incorporated to enable the service developers to perform additional activities, such as *semantics validation*, *syntax error detection*, and *code auto-completion,* where possible.

7.3.2.4. Visualization and Deployment

One challenge that commonly hinders the development of semantic services is the complexity and steep learning curve of the semantic models and the descriptions that are generated, based on such models (cf. Section 1.2 – Chapter 1). In addressing this challenge, pre-existing semantics visualization tools were incorporated within the *i*SemServ framework. The tools that were re-used include OWLViz and WSMOViz (Kerrigan, 2006), which were developed for supporting semantic service engineers with the lifecycle of semantics generation.

For the purposes of this study, these components were re-used to support the service engineer with the visualization of semantic descriptions generated using the

Semantics Descriptor module, thus making the process of reviewing and editing generated semantic descriptions and domain ontologies simpler and faster.

The deployment of generated and/or refined semantic descriptions and domain ontologies is made possible in the *i*SemServ framework by using different execution environments. As described in Section 7.3.1, syntactic descriptions are deployed in the Apache Tomcat application server, whilst semantic descriptions are either deployed in WSMX (WSMO descriptions) or Sesame (OWL-S descriptions) execution environments (cf. Section 7.2).

Multiple execution environments are used, because integrated semantic service execution environments that cater for syntactic descriptions and semantic descriptions of different models are currently lacking. Addressing this challenge is beyond the scope of this study.

7.3.3. Intelligence Layer

The agent-based principle of the *i*SemServ framework was realized within the intelligence layer. The implementation was achieved by exploiting the JADE platform[47], which provides specialized libraries for defining Java-based autonomous and social software agents.

JESS, a Java-based rule engine (Friedman-Hill, 2003), was also exploited for specifically implementing the intelligence properties (i.e. proactivity, and reactivity), according to the definition of the intelligent semantic service. However, it should be noted that the *i*SemServ framework only auto-generates the intelligent logic necessary for integrating JADE and JESS, as indicated in Listing 7.4. JESS rules are defined by the developer for the specific *IsS* using the generated JESS rules' template as shown under Appendix B (cf. JESS Rules Template)

An Acceleo transformation script (using the code structure in Listing 7.4) was implemented for auto-generating generic code for integrating JADE and JESS. The important parts in Listing 7.4 are the inclusion of JADE and JESS libraries as

[47] Visit http://jade.tilab.com for more details on JADE

indicated in Line 5 and Line 6. These are needed to enable any functionality of JESS within the intelligence layer. As it may be noted in Line 8, the JESS agent (in this case named: *iSemServJessAgent*) extends from the core JADE agent. Line 13 – 16 initializes the JESS engine for enabling automatic synthesis of domain ontologies and semantic descriptions with little human intervention.

Line 20 – 23 shows the logic that is executed during the initialization of JESS, and this code adds a basic JESS behaviour, which is auto-generated using an Acceleo script.

```
1.  /* template for implementing automated reasoning in iSemServ
    using JESS engine */
2.
3.  package isemserv.reason.jess;
4.
5.  import jess.*;
6.  import jade.core.*;
7.
8.  public class iSemServJessAgent extends jade.core.Agent
9.   {
10.
11. /* Initialization of a Jess engine */
12.
13.   private Rete jess;
14.   public Rete getRete ()
15.     {
16.     return jess;
17.     }
18. /* setup method to add basic JESS behaviour that will be used
    by the generic agents to process semantic services
19. */
20.  protected void setup() {
21.
22.     addBehaviour(new
    BasicJessBehaviour(this,"isemserv/jess/irules.clp",1));
23.   }
24. }
```

Listing 7.4: Excerpt of JESS integration with JADE[48]

In the context of this study, an Acceleo transformation script was also implemented for auto-generating generic agents using the annotations in the service model – thereby further simplifying and accelerating the process of building intelligent

[48] Adapted from examples used in JADE and JESS (see: http://jade.tilab.com and http://www.jessrules.com)

semantic services. The generic agents provide autonomy and social abilities to intelligent semantic services.

According to McIlraith, Son and Zeng (2001),*"Semantic Services need to be agent-ready, user-apparent, and machine-understandable".* Figure 7.6 depicts the generic agents (i.e. Service Provider Agent and Client Gateway), their interactions and distinct behaviours, as implemented for the suggested framework. Moreover, the interaction between the generated descriptions and their generic agents is demonstrated.

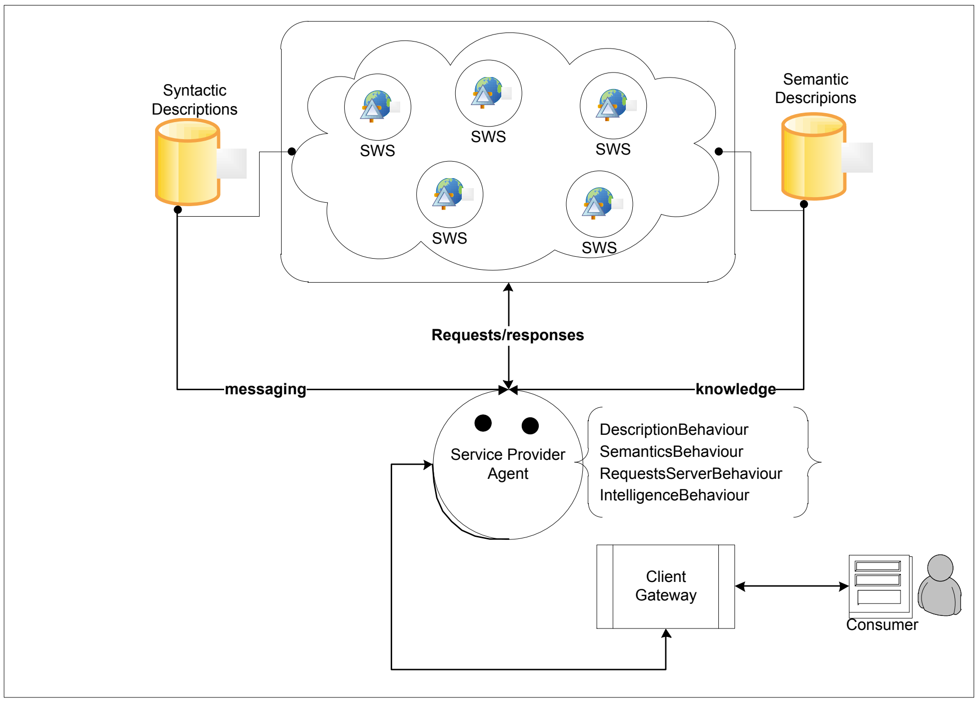


Figure 7.6: Generic Agents Interaction with SWS

The *Service Provider Agent* was implemented to be responsible for:

- **Syntactic descriptions parsing** – Autonomously parses the syntactic descriptions on behalf of the client agent to determine the functional properties of the discovered service.
- **Domain ontologies querying** – The service provider agent has the capability of querying the knowledge embedded in the semantics layer. This could, for

example, involve the task of querying a vocabulary of concepts and their relationship and axioms.

- **Semantic descriptions deduction** – This involves the automatic deduction of service functional and behavioural aspects (e.g. Web service capability), as embedded within the semantic descriptions generated in the semantics layer.
- **Intelligence behaviour implementation** – With the assistance of JESS, declarative rules are defined by the *IsS* developer to enable service consumers and providers to interact in a highly proactive and reactive manner (refer to Figure 7.7).
- **Service requests and responses management** – In this instance, every auto-generated Service Provider agent includes a service response handler, which extends JADE-based cyclic behaviour, and is generally responsible for automatically processing service requests and channelling service responses to the service consumer.

The *Service Provider Agent* also wraps the semantic service at the message-level and knowledge-level. For message-level wrapping, a generic syntactic descriptions parsing behaviour was implemented (see excerpt in Listing 7.5).

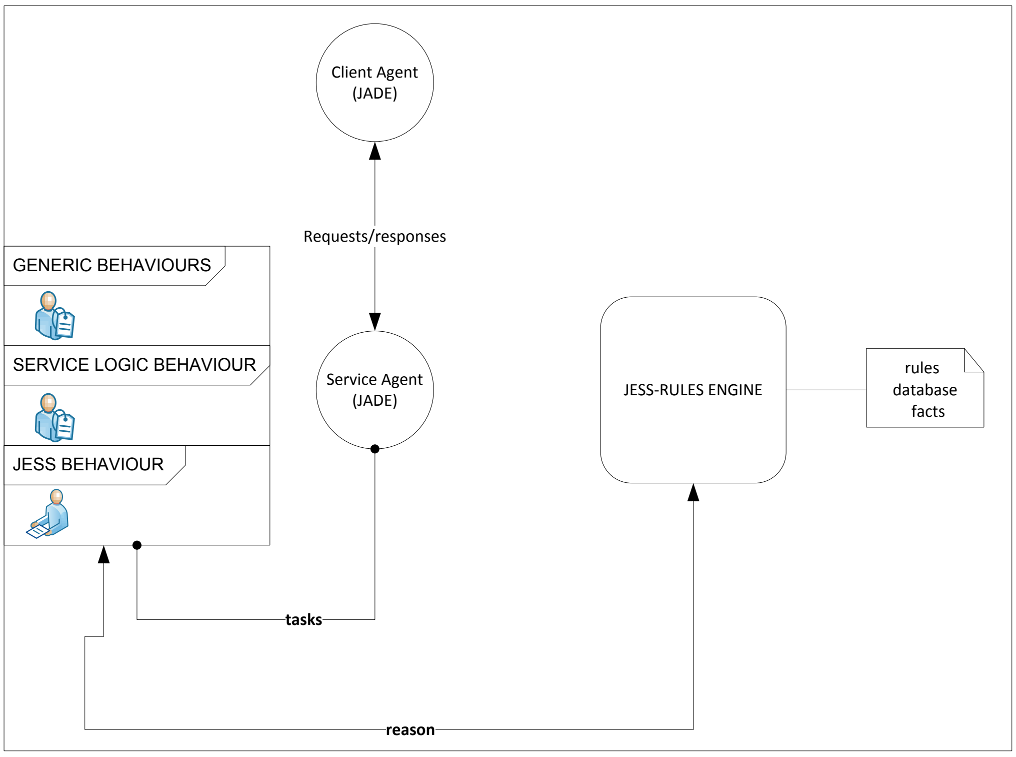


Figure 7.7: Integration of JADE agents and JESS

JADE agents understand ontologies. Thus, the *Service Provider agent* template has been implemented to accommodate the interaction between the Service and the Provider agents at the knowledge level using domain ontologies and semantic descriptions (cf. Appendix B - intelligence layer). Domain ontologies querying behaviour and semantic description parser and reasoner behaviours were also implemented – in an effort to realize intelligent semantic services.

Listing 7.5 shows the code template that was followed to create an Acceleo transformer, in order to auto-generate many of the Service Provider agent behaviours described above.

```
public class ServiceProviderAgent extends Agent {

/**
default Agent properties
*/
private static final long serialVersionUID = 1L;
private String serviceRules = "ServiceProviderRules.clp";
/**
Initialise ServiceProviderAgent
*/
protected void setup()
{
//Logging "welcome message"
System.out.println("Hallo! :"+getAID().getName()+" is initialized and ready--->");
15. /**
16. Register ServiceProviderAgent in YellowPages (JADE)
17. */
18. DFAgentDescription dfd = new DFAgentDescription();
19. dfd.setName(getAID());
20. ServiceDescription sd = new ServiceDescription();
21. sd.setType("service-provider");
22. sd.setName("iSemServ-service-provider");
23. dfd.addServices(sd);
24.
25. try {
26. DFService.register(this, dfd);
27. //Logging a confirmation  message for Registration
28. System.out.println(getAID().getName()+" is registered in JADE Yellow Pages");
29. }
30. catch (FIPAException fe) {
31. fe.printStackTrace();
32. }
33. /**
34. Add getRESTServiceURL [Part of SyntacticDescriptions Behaviour] Cyclic Behaviour provided
    by ServiceProviderAgent
35. */
36. addBehaviour(new getServiceURL());
37. /**
38. Add getWSCapabilityName [Part of SemanticsBehaviour]Cyclic Behaviour provided by
    ServiceProviderAgent
39. */
40. addBehaviour(new getWSCapability());
41. /**
42. Add the Jess Engine (Reasoner) Behaviour
43. */
44. addBehaviour( new ReasonerActivity(this, serviceRules));
45.
46. /**
47. Add the ServiceResponseHandler Behaviour
48. */
49. addBehaviour( new CompareQuotes());
50. }
```

Listing 7.5: Excerpt of the Service Provider Agent

The Service Provider agent for any generated semantic service extends the `Agent class` (see Line 1 in Listing 7.5), which is found within the JADE development libraries. The `setup()` protected method (i.e. Line 11) is used for initializing any JADE-based agent. It also implements the code necessary for registering the agent into JADE yellow pages for discovery purposes by the client agents (i.e. Line 16 - 32). The rest of the code(i.e. Line 34 - 49) is about adding respective behaviours to the service provider agent, e.g. `getWSCapability` (i.e. Line 40), which deals mainly with returning the semantic capability of the service back to the requester.

It is important to note that the generated *Service Provider agent* logic in Listing 7.5 is not static, and could be modified by the developer to suit the requirements not represented in the service model.

The *Client Gateway agent* was implemented to handle:

- **Keyword-based service discovery** – As partially illustrated in Listing 7.6, a common JADE service discovery technique was adopted for keyword-based discovery. For example, in Line 14 of Listing 7.6, we merely demonstrate how agents of type “service-provider” could be discovered using the JADE search service.

```
 1. @Override
 2. public void action()
 3. {
 4.
 5.   if(status==0)
 6.   {
 7.       //Code stubs for service discovery
 8.       DFAgentDescription template = new DFAgentDescription();
 9.       ServiceDescription sd = new ServiceDescription();
10.       sd.setType("service-provider");
11.       template.addServices(sd);
12.
13.          try {
14.          DFAgentDescription[] result = DFService.search(myAgent, template);
15.          System.out.println("Following service providers discovered:");
16.          availServiceProvider = new AID[result.length];
17.                 for (int i = 0; i < result.length; ++i) {
18.                       availServiceProvider[i] = result[i].getName();
19.
20.                       System.out.println(availServiceProvider[i].getName());
21.                 }
22.                        }
23.                        catch (FIPAException fe) {
24.                               fe.printStackTrace();
25.                        }
26.                 }
27.                 //Perform service requests
28.                 ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
29.                 for (int i = 0; i < availServiceProvider.length; ++i) {
30.                        cfp.addReceiver(availServiceProvider[i]);
31.                 }
```

Listing 7.6: Excerpt of the Client Gateway Agent

- **Service requests** – manual service requests’ user interfaces are also auto-generated within the *i*SemServ framework for each intelligent semantic service. This was implemented to simplify the process of quickly testing deployed intelligent semantic services. In this regard, the service consumer will interact with the intelligent semantic service(s) in terms of furnishing the required inputs through a Web-user interface (e.g. PHP web page with a form).

7.3.4. Front-end

Although the *i*SemServ framework and architecture is divided into various layers with various modules, the front-end system representing the implemented platform is only an Eclipse plug-in that could easily be integrated within version 3.5 of Eclipse Galileo. The user interface that the developer interacts with is depicted in Figure 7.8.

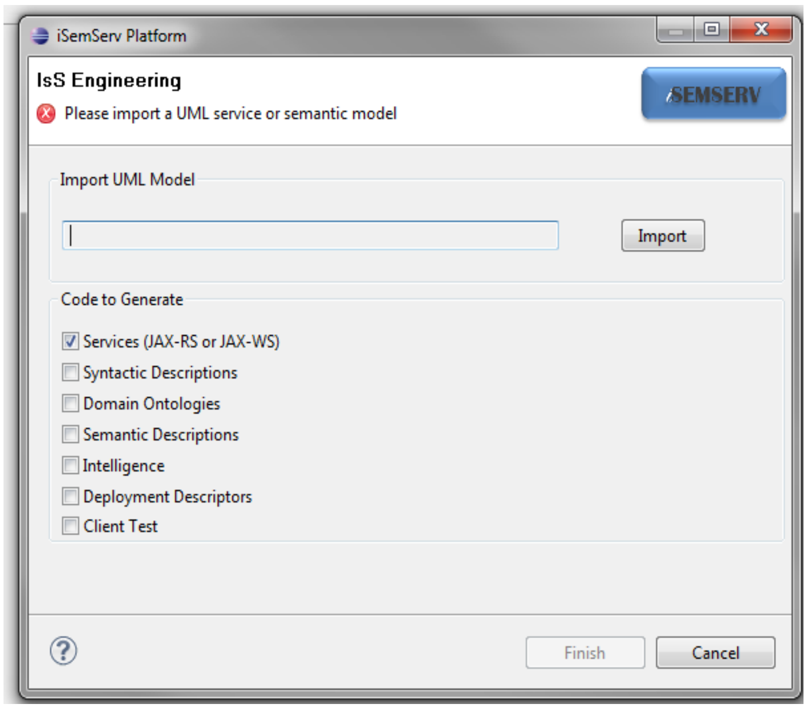
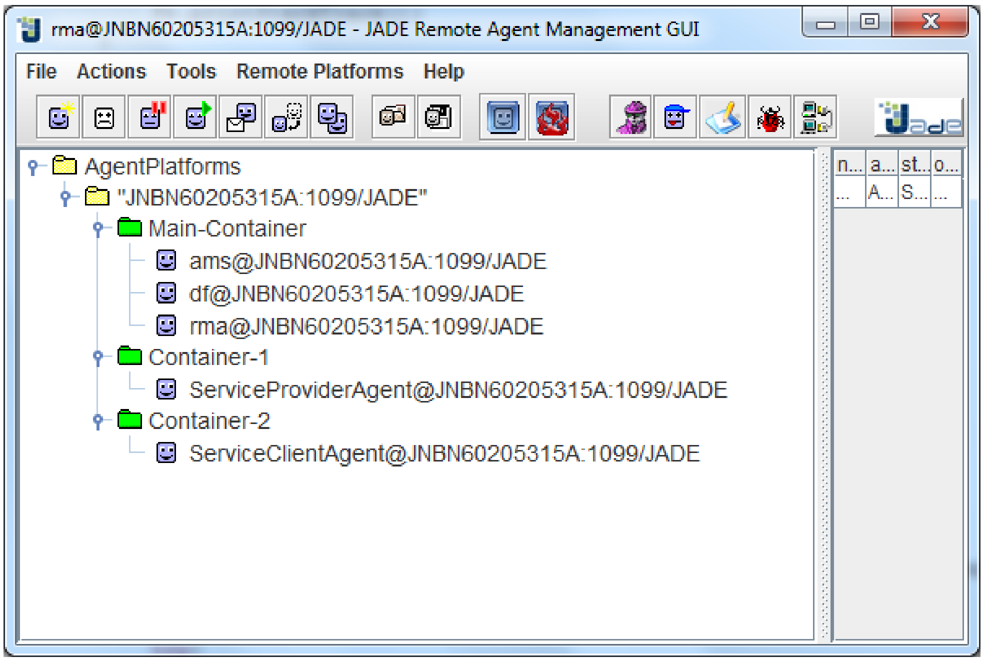


Figure 7.8: *i*SemServ Eclipse Plug-in

The service engineer only needs to import a UML2 packaged service model, and select the types of service elements that need to be auto-generated. The plug-in would then in the background generate all the selected service elements using the defined transformation rules and templates. The engineer could then review, edit, and finalize the generated modules or even re-use previously generated elements (e.g. domain ontologies) using various editors integrated within the Eclipse environment, as described above.


rma@JNBN60205315A:1099/JADE - JADE Remote Agent Management GUI
File Actions Tools Remote Platforms Help
n... a... st... o...
... A... S... ...

```
AgentPlatforms
  "JNBN60205315A:1099/JADE"
    Main-Container
      ams@JNBN60205315A:1099/JADE
      df@JNBN60205315A:1099/JADE
      rma@JNBN60205315A:1099/JADE
    Container-1
      ServiceProviderAgent@JNBN60205315A:1099/JADE
    Container-2
      ServiceClientAgent@JNBN60205315A:1099/JADE
```

Figure 7.9: JADE runtime environment

As illustrated in Figure 7.9, the generated agent artefacts are deployed within the containers, as supported by the JADE platform. Nevertheless, the deployment of intelligent semantic services falls outside the scope of this thesis. The actual operation of intelligent semantic services, as generated within the *i*SemServ framework, is demonstrated with the aid of a use case scenario presented in Chapter 8.

7.4. SUMMARY

The *i*SemServ framework implementation details on the Eclipse platform were presented in this chapter. The framework was implemented according to its defined layers. These are: services, semantics, and intelligence layer. In implementing the framework, a number of technological tools were used, and these were also discussed in this chapter. The overall implementation was realized by re-using a number of already available open source and limited proprietary tools. The key technologies used to implement the different modules of different layers include the UML2 SDK, which was used for designing service models. Java as an implementation language was also used throughout the different layers. Only RESTful and SOAP-based services have been accommodated within the proposed framework.

In terms of semantic descriptions and domain knowledge, WSMO and OWL-S modules were implemented. The core contribution of this study, which is the intelligence layer, was implemented using the JADE and JESS environments, in order to realize autonomous, social, reactive, and proactive features that make semantic services intelligent. Acceleo was used throughout the different layers for the purpose of defining templates and transformation rules for auto-generating different service elements.

A number of lessons were learnt during the development of the prototype. These included the understanding that developing intelligent semantic services is indeed a tedious and an error-prone task – especially without any supporting tools. We envisage the suggested approach as one possibility for minimizing the current hindrances of engineering intelligent semantic services, especially because of the principles of standards independency, separation of concerns, complexity hiding, and the exploitation of existing technologies, including the integration of semantic technologies into existing web services platforms, such as Eclipse.

Nevertheless, the proposed *i*SemServ framework also has its own challenges. These are discussed in Chapter 9.

The following chapter discusses the assessment of the use case scenarios, as well as the evaluation of the proposed and implemented solution.

CHAPTER 8: Evaluation and Results

This chapter demonstrates the use, together with the evaluation results of the proposed *i*SemServ framework. The framework was evaluated using different techniques including a qualitative comparison of the semantic services engineering platforms and real-world use case scenarios. These are discussed in this chapter.

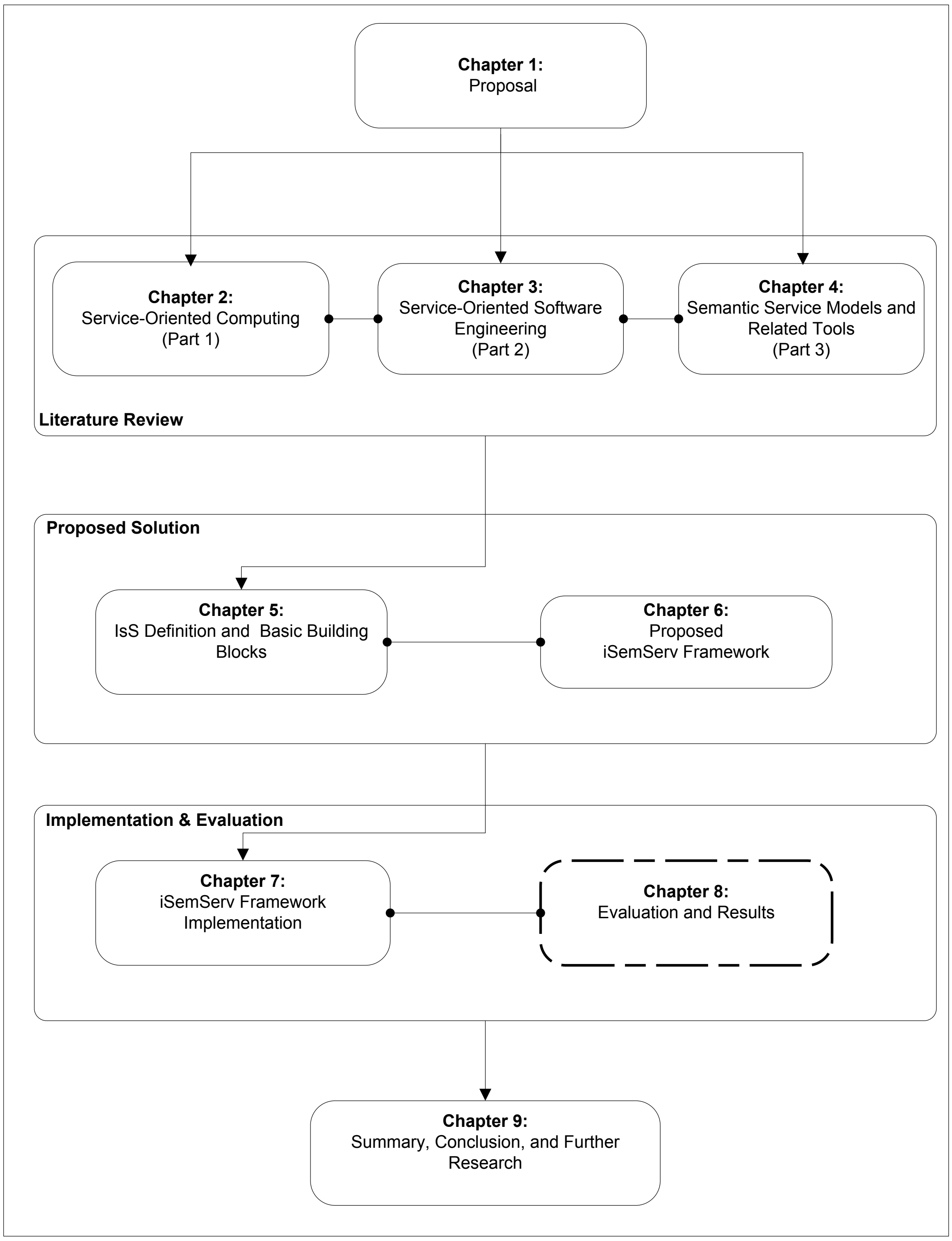

Chapter 1: Proposal
Chapter 2: Service-Oriented Computing (Part 1)
Chapter 3: Service-Oriented Software Engineering (Part 2)
Chapter 4: Semantic Service Models and Related Tools (Part 3)
Literature Review
Proposed Solution
Chapter 5: IsS Definition and Basic Building Blocks
Chapter 6: Proposed iSemServ Framework
Implementation & Evaluation
Chapter 7: iSemServ Framework Implementation
Chapter 8: Evaluation and Results
Chapter 9: Summary, Conclusion, and Further Research

Figure 8.1: Overall Thesis Structure

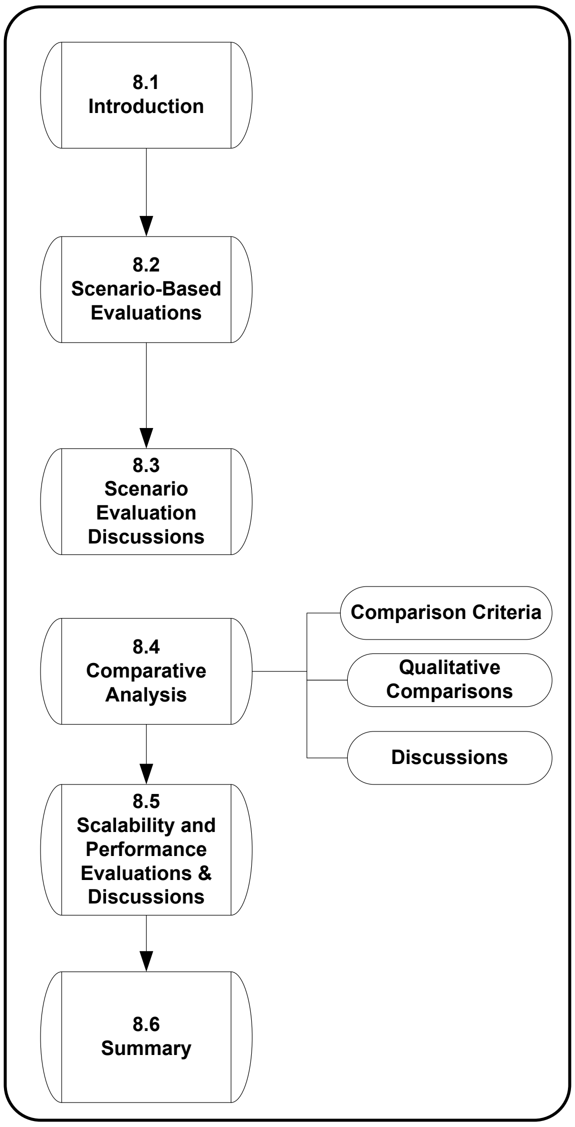


Figure 8.2: Chapter 8 Layout

8.1. INTRODUCTION

One of the additional objectives of this thesis is to evaluate the proposed service creation framework against the design principles set out in Chapter 6, related solutions as presented in Chapter 2, and appropriate use case scenarios, as defined in Section 8.2. Thus, this chapter discusses the evaluation of the *i*SemServ framework, as implemented in Chapter 7. The framework is evaluated using qualitative comparative analysis, laboratory experiments, and quantitative performance and scalability tests.

In order to reach reliable and valid conclusions in relation to the proposed framework and its implementation, the evaluation process plays an important part. Moreover, evaluation is essential to the development of any technical solution. However, approaches for evaluating unified semantic-based solutions are not currently well-established or standardized. Thus, in evaluating the *i*SemServ framework and its implementation, different types of evaluation techniques were considered. The initial evaluation focused on real-world use case scenarios, which were specifically defined for Semantic Web Services (SWS).

Real-world scenarios are key in assessing technical solutions (Kuropka *et al.*, 2008). Scenarios could also enable an evaluation of the different aspects of the technical solution, such as the design, functionality, scalability, and performance. In the context of this thesis, a number of real-world and mature scenarios were adopted, and prototyped – to demonstrate the operations of the *i*SemServ environment. Furthermore, controlled laboratory experiments were also carried out to illustrate the practicality and relevance of the proposed solution in relation to the engineering effort involved in building intelligent semantic services.

In addition, comparative analysis (Hofstee, 2006) plays an important role in assessing any new solution against the existing similar solutions. As a result, for evaluating the *i*SemServ framework and its implementation, a comparative analysis was also conducted, in order to qualitatively note the benefits and the limitations of the proposed solution against the common related solutions.

The remainder of this chapter is structured as follows: Scenario-based evaluations are introduced and discussed in Section 8.2. The evaluation results from the experiments are discussed in Section 8.3. In Section 8.4, a comparative analysis using the *i*SemServ framework design principles as a base is discussed.

Adopting the SEALS[49] methodology for evaluating semantic web services tools, scalability and performance tests of the *i*SemServ platform are performed; and the results are presented in Section 8.5. The chapter is concluded with a summary of the evaluations in Section 8.6.

8.2. SCENARIO-BASED EVALUATIONS

As mentioned throughout this thesis, the main goal of the proposed *i*SemServ framework is to simplify and accelerate the process of engineering intelligent semantic services. In assessing the applicability and benefits of the implemented solution, particularly with regard to satisfying the main design principles (i.e. simplification and acceleration), existing real-world use case scenarios were adopted from the European Union (EU) framework projects, such as DIP (Data, Information and Process Integration with Semantic Web Services) (Losada *et al.*, 2005), and SWSA (Semantic Web Services Architecture) usage scenarios[50].

These real-world scenarios were adopted, as they are well defined for semantic services' environments, and have been implemented in different ways in the aforementioned projects. In addition, real-life project scenarios were adapted and experimented to evaluate the scalability and the performance of the *i*SemServ solution.

Although the objectives of the selected scenarios were mainly to demonstrate the relevance of semantic services in the world of Web services, in the context of this thesis, the scenarios were further adapted to demonstrate the processes of engineering intelligent semantic services. In line with the objectives of this study, our focus on the scenarios was about demonstrating the benefits of the *i*SemServ

[49] See: http://www.seals-project.eu
[50] SWSA scenarios are available at: http://www.ai.sri.com/daml/services/use-cases.html

framework, rather than demonstrating working service-oriented applications. The following sub-section presents one of the several experimented use case scenarios.

8.2.1. Online Multimedia Trading

Traditional Web services do not provide explicit semantic representations during service requests and responses. This usually leads to a number of issues – as was discussed in Chapter 1, some of which include ambiguous interpretations of service operations and inconsistent or unreliable service responses.

The scenario presented in this section demonstrates how intelligent semantic services could be implemented in a simple and efficient manner by using the *i*SemServ plug-in. This real-world scenario involves tasks that have been assigned to the service developer. The tasks involve developing an online multimedia trading Web application that enables consumers and sellers to perform the following activities, using intelligent semantic services:

- Search for different multimedia products in a semantically-enabled multimedia catalogue;
- Dynamically add multimedia products to the shopping cart;
- Order products in the shopping cart;
- Use external services to make payments;
- Intelligently add new multimedia products to the catalogue.

The requirement is also to implement RESTful services grounded in WSMO ontologies. Existing domain ontologies describing multimedia products, such as those from Amazon[51] could also be exploited. This scenario, as depicted in Figure 8.3, is adapted from the Amazon use case of selling and buying books online, using Web services, as defined by SWSA.

[51] See: http://www.wsmo.org/ontologies/amazonECS/amazonOntology.wsml

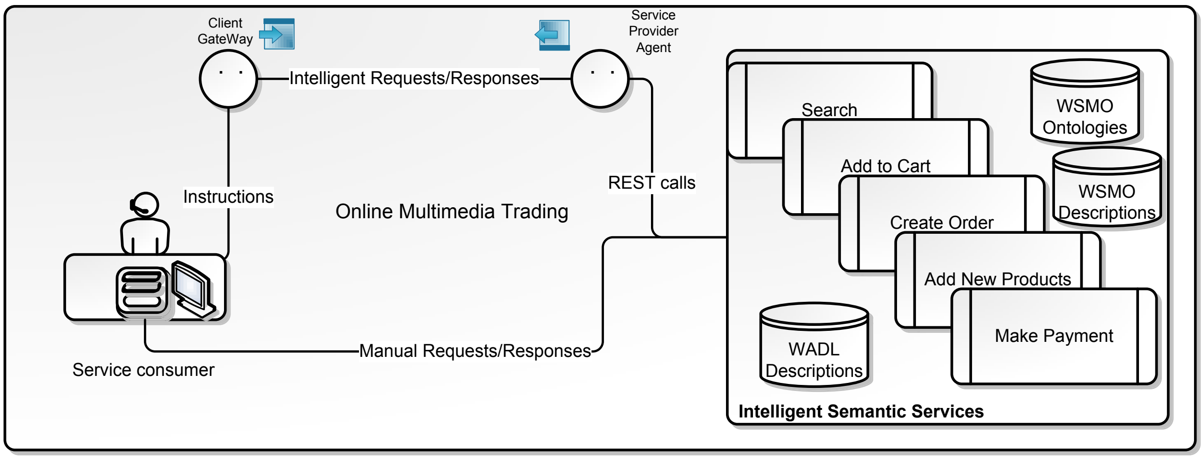
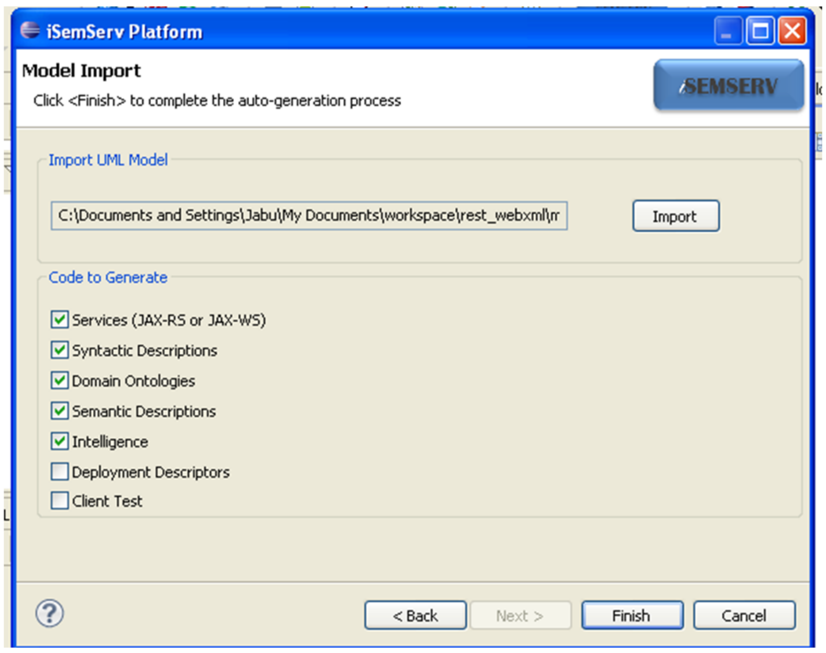


Figure 8.3: Online Multimedia Trading Scenario

The following section illustrates how the above-mentioned scenario could be engineered by the service developer using the proposed Eclipse plug-in, as demonstrated in Figure 8.4.



Model Import

Click <Finish> to complete the auto-generation process

Import UML Model

C:\Documents and Settings\Jabu\My Documents\workspace\rest_webxml\r

Code to Generate

- [✓] Services (JAX-RS or JAX-WS)
- [✓] Syntactic Descriptions
- [✓] Domain Ontologies
- [✓] Semantic Descriptions
- [✓] Intelligence
- [ ] Deployment Descriptors
- [ ] Client Test

Figure 8.4: *i*SemServ Plug-in

8.2.1.1. Service models

The service developer uses the UML2 SDK plugged into Eclipse to design service models capturing both the services' structures and semantic concepts. It is further emphasized that the model is defined according to the *i*SemServ UML profile presented in Chapter 7 (Section 7.3). Figure 8.5 depicts the service model for the online multimedia trading scenario.

This model is inputted into the plug-in through the simple browser capability. The developer chooses the modules that need to be generated. The *i*SemServ transformation module then automatically generates the selected implementation artefacts, which are described in the following sections.

8.2.1.2. Syntactic Web services

As may be noted, 6 classes are modelled and annotated with appropriate keywords (e.g. **<<RESTful>>** and **<<WSMO>>**). From the service model, syntactic RESTful services are generated according to the **<<RESTful>>** annotation. In this regard, the *i*SemServ environment facilitates the generation of skeleton syntactic RESTful services.

The service developer would be responsible for completing the service logic using the generated skeleton classes. At this layer, simplification and acceleration of the engineering process is addressed through automatic code generations.

The amount of time it takes, for example, to generate the skeleton code for the classes depicted in the model is only a few milliseconds (cf. Section 8.5) compared with manually coding the structure of RESTful services. However, this is not novel, as this methodology is used extensively in a number of mature development environments, such as Eclipse and Visual Studio. The key difference is that in the *i*SemServ platform, the service developer is in control of what code skeletons could be generated through the use of service models and profiles.

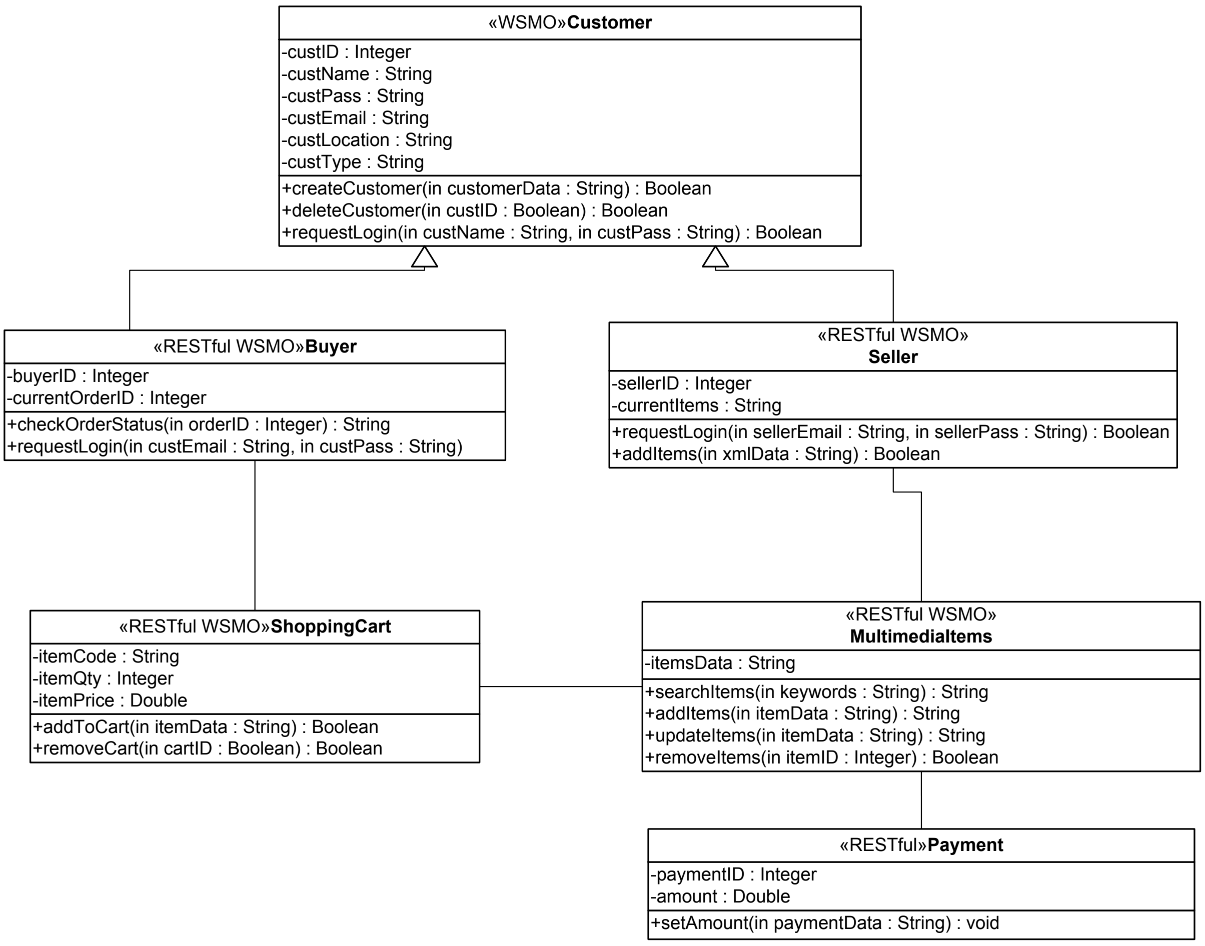

«WSMO»Customer
-custID : Integer
-custName : String
-custPass : String
-custEmail : String
-custLocation : String
-custType : String
+createCustomer(in customerData : String) : Boolean
+deleteCustomer(in custID : Boolean) : Boolean
+requestLogin(in custName : String, in custPass : String) : Boolean
«RESTful WSMO»Buyer
-buyerID : Integer
-currentOrderID : Integer
+checkOrderStatus(in orderID : Integer) : String
+requestLogin(in custEmail : String, in custPass : String)
«RESTful WSMO» Seller
-sellerID : Integer
-currentItems : String
+requestLogin(in sellerEmail : String, in sellerPass : String) : Boolean
+addItems(in xmlData : String) : Boolean
«RESTful WSMO»ShoppingCart
-itemCode : String
-itemQty : Integer
-itemPrice : Double
+addToCart(in itemData : String) : Boolean
+removeCart(in cartID : Boolean) : Boolean
«RESTful WSMO» MultimediaItems
-itemsData : String
+searchItems(in keywords : String) : String
+addItems(in itemData : String) : String
+updateItems(in itemData : String) : String
+removeItems(in itemID : Integer) : Boolean
«RESTful»Payment
-paymentID : Integer
-amount : Double
+setAmount(in paymentData : String) : void

Figure 8.5: Online Multimedia Trading Service Model

A snippet of the generated code structure is shown in Figure 8.6. This structure demonstrates the number of classes (i.e. six) generated based on the number of classes modelled in UML.

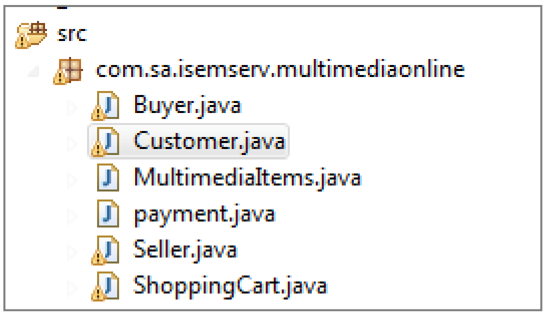

src
com.sa.isemserv.multimediaonline
Buyer.java
Customer.java
MultimediaItems.java
payment.java
Seller.java
ShoppingCart.java

Figure 8.6: Syntactic RESTful Services

As illustrated in the service model (cf. Figure 8.5), five UML classes represent five RESTful services, while one, that is, the `Customer` class, is not a RESTful service, but a pure Java POJO class. Nevertheless, semantic descriptions and domain ontologies for this class are also generated on the basis of the **<<WSMO>>** annotation.

The excerpt of the skeleton code behind RESTful services (e.g. Seller) is illustrated in Listing 8.1. This shows the code auto-generated based on the service model. Line 5 indicates that the generated code represent a RESTful service (JAX-RS); as a result of the **@Path** annotation (cf. Chapter 7, Table 7.2).

```
1.  /**
2.        @Path
3.       represents relative URI for a RESTful resource
4.  */

5.  @Path("/seller")
6.  public class Seller extends Customer  {
7.  /*
           @Declaration of Attributes
8.  */
9.  private int sellerID;
10. public String currentItems;
11. /*
           @Declaration of Operations
12. */
13. /*
           Description of the method requestLogin
14. *
           @param CustEmail
           @param custPass
           @return Boolean
15. */
16. /**
           decorate our RESTful service with @Path, @HTTP_Method, and @Representation
17. */
18. @Path("/requestlogin")
19. @GET
20. @Consumes({"text/plain","application/xml","text/html","application/json"})
21. @Produces({"text/plain","application/xml","text/html","application/json"})
22. public Boolean requestLogin(@PathParam("CustEmail custPass")String CustEmail,String
    custPass){

23. //TODO: ADD service logic for requestLogin method

24. return null;
25. }
```

Listing 8.1: Seller RESTful Skeleton Code

A RESTful method is illustrated by the **@GET** annotation (Line 19), and other mappings as discussed in Chapter 7 (cf. Section 7.3, Table 7.1). As may be noted in Line 23, the developer would then need to add the service logic for the generated method. The developer could also edit the generated code in whatever way that is deemed necessary.

8.2.1.3. Syntactic descriptions

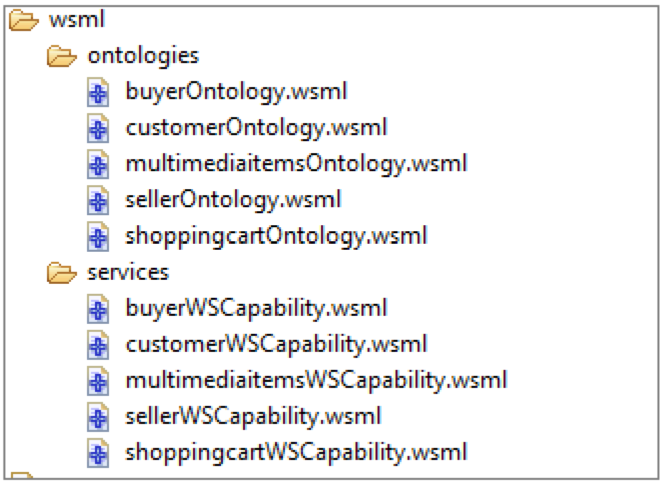
As highlighted throughout this thesis, the key benefit of Web services is that they are self-described for the purposes of discovery, selection, and manual composition. Thus, the *i*SemServ platform also makes it possible to auto-generate WADL descriptions for every RESTful class or service. It should be noted that WADL descriptions can be auto-generated in two ways, (1) *annotating the model with the* ***<<WADL>>*** *stereotype,* and (2) *using the* ***<<RESTful>>*** *annotation and choosing the semantic descriptions option on the iSemServ plug-in*. An example of the WADL descriptions generated for the online multimedia scenario is illustrated in Listing 8.2. The listing only represents the syntactic description for multimedia items class. The auto-generated descriptions are linked to the mapping rules in Chapter 7 (cf. Section 7.3, Table 7.3).

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- Generated by SemServ Model2Descriptions transformer using Acceleo 2.8 -->
<!-- Date: May 25, 2012 [11:19:39 AM] -->
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://wadl.dev.java.net/2009/02 wadl.xsd"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://wadl.dev.java.net/2009/02">
<doc xmlns:semserv="http://desc.isemserv.co.za/" />
<doc xml:lang="en" title="Documentation for MultimediaItemsService">
documentation for  application.wadl
</doc>
<grammars>
<include href="{add reference to schemas if any}"/>
</grammars>
<resources base="http://localhost:8088/restful">
<resource path="/multimediaitems">
<resource path="/update">
<method name="POST" id="updateitems">
<request>
<representation mediaType="application/xml"/>
</request>
<response>
<representation mediaType="application/text"/>
</response>
</method>
</resource>
</resource>
</resources>
</application>
```

Listing 8.2: Partial WADL Description

8.2.1.4. Semantic descriptions

Semantic descriptions and domain ontologies are dynamic, in a sense that they evolve over time. As a result, they are resource intensive to build and update. In Figure 8.7, the snippet of semantic descriptions and domain ontologies auto-generated for each class annotated with **<<WSMO>>** stereotype are shown. The generation process also depends on the service model presented in Figure 8.5. The semantic descriptions at this phase are independent of the syntactic descriptions discussed in Section 8.2.1.3.



```
wsml
    ontologies
        buyerOntology.wsml
        customerOntology.wsml
        multimediaitemsOntology.wsml
        sellerOntology.wsml
        shoppingcartOntology.wsml
    services
        buyerWSCapability.wsml
        customerWSCapability.wsml
        multimediaitemsWSCapability.wsml
        sellerWSCapability.wsml
        shoppingcartWSCapability.wsml
```

Figure 8.7: Generated Semantic Descriptions

Basically, the `[`*`ClassName`*`]+WSCapability.wsml` files represent Web Service capabilities according to WSMO specifications. As may be noted in Figure 8.7, the scenario in question is semantically described with five domain ontologies, and five Web service capabilities, referred to as semantic descriptions throughout this study.

The generated WS capability skeleton code for the `CheckOrderStatus` and `RequestLogin` operations is demonstrated in Listing 8.3. The service engineer could further edit the generated code using semantic tools, such as the WSMO editor embedded within the Eclipse environment. The code is generated based on the mapping rules discussed in Chapter 7 (cf. Section 7.3, Table 7.5 and Table 7.7).

```
1.  wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"
2.  comment <!--Generated by SemServ Model2Semantics transformer using Acceleo 2.8-->
3.  comment <!--Date: May 25, 2012 [11:19:42 AM] -->
4.  namespace { _"http://www.isemserv.co.za/services/buyerSemantics#",
5.  buy _"http://www.isemserv.co.za/ontologies#",
6.  dc _"http://purl.org/dc/elements/1.1#",
7.  wsml _"http://www.wsmo.org/wsml/wsml-syntax#",
8.  xsd  _"http://www.w3.org/2001/XMLSchema#",
9.  desc _"http://www.isemserv.co.za/descriptions#"}

10. webService checkOrderStatusrequestLoginService

11. importsOntology {_"http://example.org/ImportedOntology"}  /*PLEASE COMPLETE*/

12. capability checkOrderStatusrequestLoginCapability

13. nonFunctionalProperties
14. dc#typehasValue"service ontology"
15. dc#descriptionhasValue"Enter description for this capability"
16. dc#titlehasValue"Capability for a buyer Web service"
17. dc#creatorhasValue {"Your Name"}
18. dc#publisherhasValue"isemserv"
19. dc#datehasValue"May 25, 2012 [11:19:42 AM]"
20. dc#typehasValue _"http://www.wsmo.org/2004/d2#ontologies"
21. dc#identifierhasValue _"http://www.isemserv.co.za/services/buyer"
22. dc#languagehasValue"en-US"
23. dc#formathasValue"text/plain"
24. desc#serviceDescription hasValue"COMPLETE URL FOR SERVICE DESCRIPTION"
25. endNonFunctionalProperties

26. sharedVariables {?orderID}
27. sharedVariables {?custEmail, ?custPass}

28. precondition
29. nonFunctionalProperties
30. dc#descriptionhasValue"condition(s) that need to be satisfied before service is
    invoked"
31. endNonFunctionalProperties
32. definedBy
33. ?orderID memberOf OrderID
34. ?custEmail memberOf CustEmail
35. ?custPass memberOf CustPass
```

Listing 8.3: Partial WSMO Service Capability

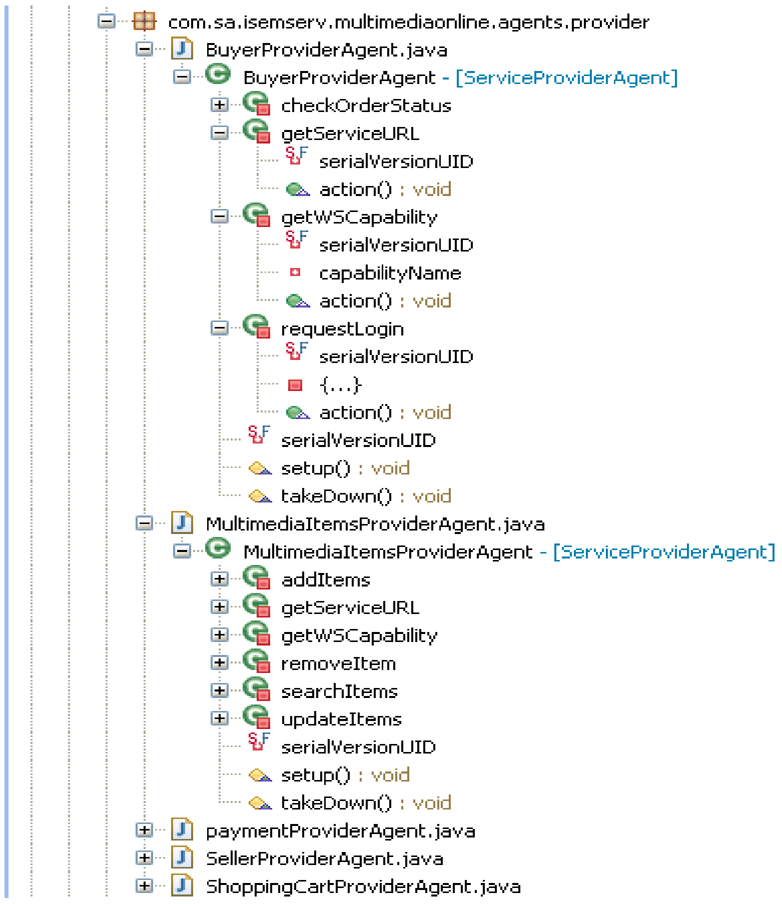
As may be noted in Listing 8.3, Line 10-Line 25, illustrate the auto-generated semantic descriptions (e.g. service name, capability name, and non-functional properties) - necessary for semantically enabling the buyer RESTful service. Nevertheless, the developer could change the descriptions to suit own development requirements. In addition, Line 11 indicates that external ontologies can also be referenced for purposes of augmenting the generated descriptions. The rest of the code is directly linked to the mappings presented in Chapter 7 (cf. Section 7.3).

8.2.1.5. Service agents

In order to enable the semantic services generated above to have the intelligent features, as described throughout this thesis, the *i*SemServ environment also enables the generation of the *Service Provider agent* and *Client agent* code that would make it possible to consume the semantic services with minimal user intervention. The service engineer would still be able to manually call generated semantic services without relying on generated provider agents for service discovery and consumption.

For the multimedia trading scenario, intelligent agents and JESS rules' template are auto-generated. The generated agents (i.e. service provider agents) for each semantic service are by default endowed with intelligent capabilities, such as keyword service discovery, syntactic descriptions parsing, semantic descriptions analysis, and service request and response management. Snippets of some of these key capabilities are highlighted in Figure 8.8.

The service engineer would only be responsible for implementing the logic of specific services, such as `checkOrderStatus`.



```
com.sa.isemserv.multimediaonline.agents.provider
    BuyerProviderAgent.java
        BuyerProviderAgent - [ServiceProviderAgent]
            checkOrderStatus
            getServiceURL
                serialVersionUID
                action() : void
            getWSCapability
                serialVersionUID
                capabilityName
                action() : void
            requestLogin
                serialVersionUID
                {...}
                action() : void
            serialVersionUID
            setup() : void
            takeDown() : void
    MultimediaItemsProviderAgent.java
        MultimediaItemsProviderAgent - [ServiceProviderAgent]
            addItems
            getServiceURL
            getWSCapability
            removeItem
            searchItems
            updateItems
            serialVersionUID
            setup() : void
            takeDown() : void
    paymentProviderAgent.java
    SellerProviderAgent.java
    ShoppingCartProviderAgent.java
```

Figure 8.8: Skeleton Code Structure for Provider Agents

The mapping of syntactic descriptions to generated provider agents at the message level, and of semantic descriptions and generated agents at the knowledge level are realized during the process of model transformation – using the code structure and templates discussed in Chapter 7 (cf. Section 7.4, Listing 7.4). In additional, essential operations, such as `getWSCapability` in Listing 8.4 and others, make such mapping possible.

```
1.  //the client wants to know the semantic capability of the service
2.  if (request.equalsIgnoreCase("getWSCapability"))
3.  {
4.  //a service response reply
5.  ACLMessage reply = msg.createReply();

6.  try {
7.  capabilityName = new WSMLReader().returnCapability();
8.  } catch (IOException e) {

9.  e.printStackTrace();
10. }

11. // The service response
12. reply.setPerformative(ACLMessage.INFORM);
13. reply.setContent(capabilityName);
14. reply.addReceiver( msg.getSender() );
15. myAgent.send(reply);
16. }
```

Listing 8.4: Semantics and Provider Agent Mapping

The code to note in Listing 8.4 is in Line 2, which is responsible for receiving a capability request message from the client agent to determine the capabilities of the discovered service. Once the message is received and is interpreted accordingly, the provider agent will respond with the functional capabilities of the service as described in the semantic descriptions (cf. Listing 8.3). As it may be noted, the agents use the Agent Communication Language (ACL) for exchanging messages (cf. Line 12 – Line 15).

The *i*SemServ framework, as highlighted in Chapter 7, does not auto-generate JESS rules for each specific *IsS*. But, the logic that integrates JADE and JESS as demonstrated in Chapter 7 (Listing 7.4), and the template for defining the specific rules. A sample of JESS rules were defined for the scenario demonstrated in Figure 8.3. Some of the rules are shown in Listing 8.5.

The first part of Listing 8.5 shows the generic template that is used for declaring a template that is needed in JESS for defining rules. The template is declared using the keyword `deftemplate` as shown in Line 1 (Part 1). The declaration includes a name that is by convention similar to the class name (e.g. Buyer cf. Figure 8.5). The template takes the same format as the corresponding class found in the UML model (Line 2). In the implementation of the *i*SemServ framework, this template is auto-generated, further simplifying the process of building intelligent semantic services.

Generated JESS rule template (Part 1)

```
1  (deftemplate buyer
2    (declare (from-class buyer)
3    (include-variables TRUE))
```

JESS rule template (Part 2)

```
1  "This rule automatically processes the order ID"
2
3  (defrule processOrderID
4     "Processing Order"
5      (buyer {OrderID < 1})
6   )
7      =>
8      (assert (buyer (OrderID 1)))
9      (printout t "Order successfully placed" crlf))
10
11 "check automatically if order was placed, assuming that if
12 OrderID is greater than 1 it means order is placed"
13
14 (defrule check-if-order-placed
15    "Checking Order"
16     (buyer {orderID > 0})
17     =>
18     (printout t "Order was placed", crlf)
19  )
20
```

Listing 8.5: Excerpt of JESS template and rules

Part 2 of Listing 8.5 include some of the rules that were manually defined using JESS programming environment integrated within Eclipse. The rules are based on the generated template in Part 1. The first rule (Line 3 – Line 9) enables the Provider agent to process the `OrderID` when a specific order has been successfully placed. The second rule is tied to the first rule. It enables an agent to automatically check if an order has been placed successfully by checking the value of the `OrderID`.

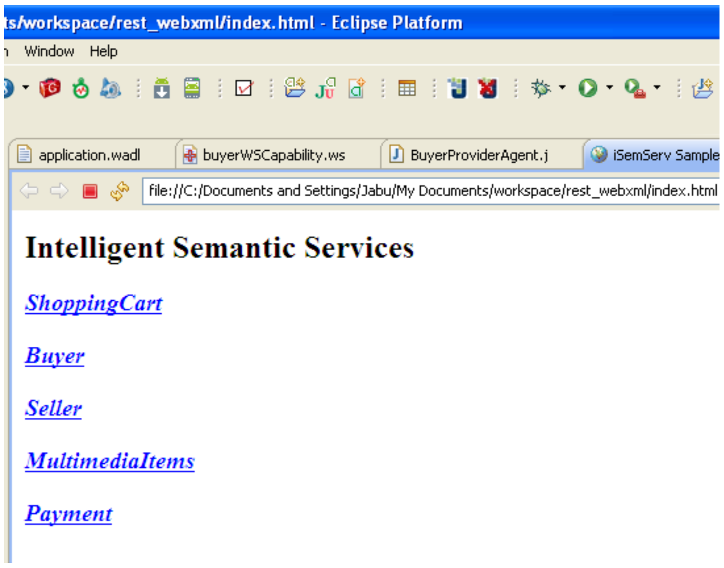
As it may be noted, JESS rules can be simple. However, for real-life applications, complex rules are unavoidable, and the developer would still need to manually define them for each specific intelligent semantic service. As highlighted, the *i*SemServ framework in its current form does not accommodate the auto-generation of JESS rules.

Once the service engineer has implemented the additional logic and defined the associated JESS rules based on the auto-generated templates for different provider agents, the online multimedia trading application can be deployed, using auto-generated deployment descriptors, and tested using generated client tests' user interfaces, as briefly discussed in the next section.

8.2.1.6. User interfaces Generation

The *i*SemServ framework further simplifies and accelerates the process of engineering intelligent semantic services by auto-generating optional Web-based user interfaces (UIs) for testing the operations of all generated intelligent semantic services. The auto-generation of UIs is a well-studied subject (Dannecker *et al.*, 2010).

The auto-generated UIs are based on Web technologies, such as HTML, JavaScript's, and Servlets. These UIs are also generated from the service models, using the Acceleo and simple Web application templates. Examples of UIs for some of the auto-generated services are demonstrated in Figure 8.9 and Figure 8.10.


ts/workspace/rest_webxml/index.html - Eclipse Platform

file://C:/Documents and Settings/Jabu/My Documents/workspace/rest_webxml/index.html

Intelligent Semantic Services

ShoppingCart

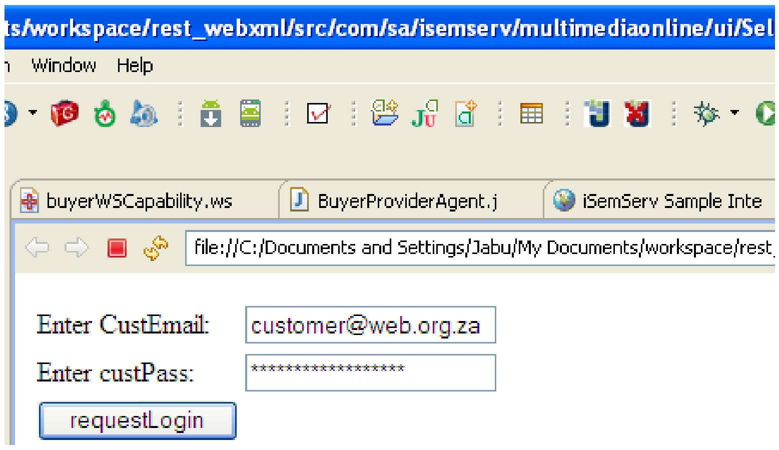
Buyer

Seller

MultimediaItems

Payment

Figure 8.9: Test User Interfaces


ts/workspace/rest_webxml/src/com/sa/isemserv/multimediaonline/ui/Sel

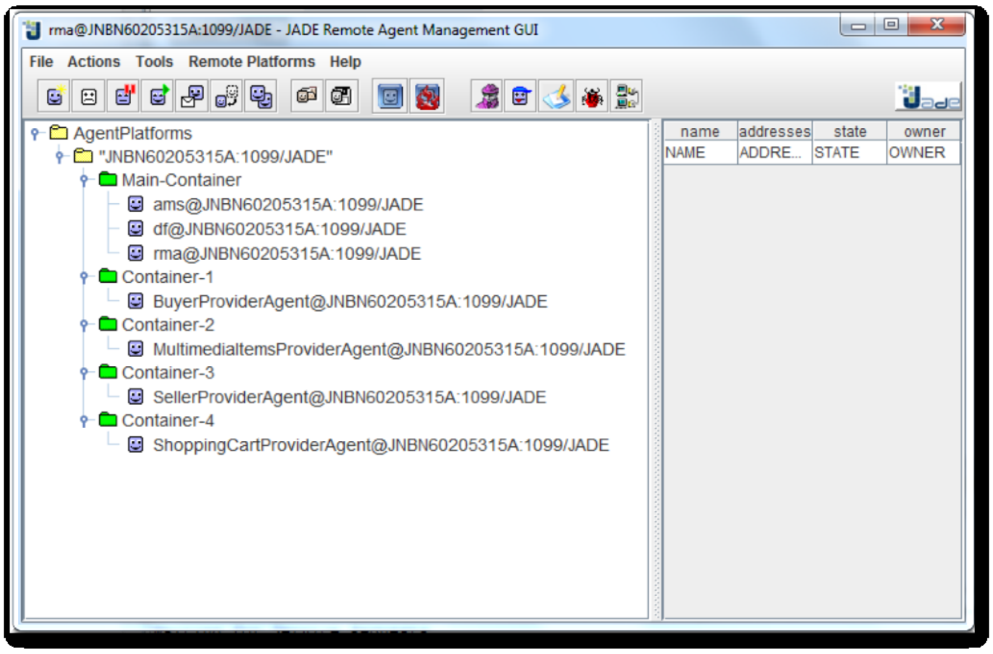
file://C:/Documents and Settings/Jabu/My Documents/workspace/rest

Enter CustEmail: customer@web.org.za

Enter custPass: ******************

Figure 8.10: A Simple Form for Testing Services

All auto-generated provider and client agents for each semantic service can be deployed to the JADE runtime environment, as illustrated in Figure 8.11. These agents would then collaborate by automatically processing semantic service requests and responses in the background.


rma@JNBN60205315A:1099/JADE - JADE Remote Agent Management GUI
File
Actions
Tools
Remote Platforms
Help

```
AgentPlatforms
  "JNBN60205315A:1099/JADE"
    Main-Container
      ams@JNBN60205315A:1099/JADE
      df@JNBN60205315A:1099/JADE
      rma@JNBN60205315A:1099/JADE
    Container-1
      BuyerProviderAgent@JNBN60205315A:1099/JADE
    Container-2
      MultimediaItemsProviderAgent@JNBN60205315A:1099/JADE
    Container-3
      SellerProviderAgent@JNBN60205315A:1099/JADE
    Container-4
      ShoppingCartProviderAgent@JNBN60205315A:1099/JADE
```

name	addresses	state	owner
NAME	ADDRE...	STATE	OWNER

Figure 8.11: JADE Runtime Environment (Provider Agents Running)

Moreover, during the implementation of different artefacts (e.g. descriptions and ontologies), the *i*SemServ environment integrates effectively with other external engineering tools for the purposes of augmenting auto-generated artefacts. These include, for instance, visualization tools, such as WSMOViz that enables service developers to visually analyze auto-generated semantic descriptions and domain ontologies.

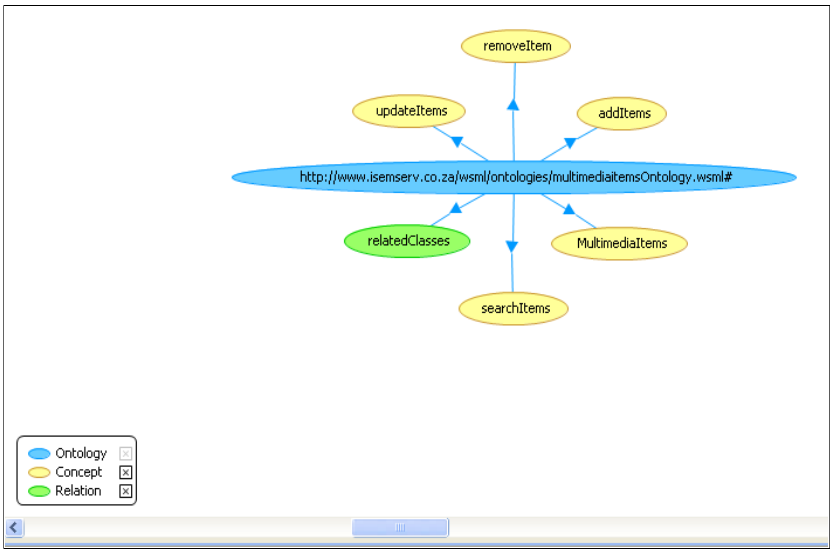


Figure 8.12: Multimedia Items Ontology Visualization

In Figure 8.12, the auto-generated multimedia-items domain ontology is shown using the WSMO Visualizer integrated in Eclipse. Moreover, Figure 8.13 depicts a tree of all concepts, relations, and axioms used in the Amazon Ontology that was imported for the online multimedia scenario.

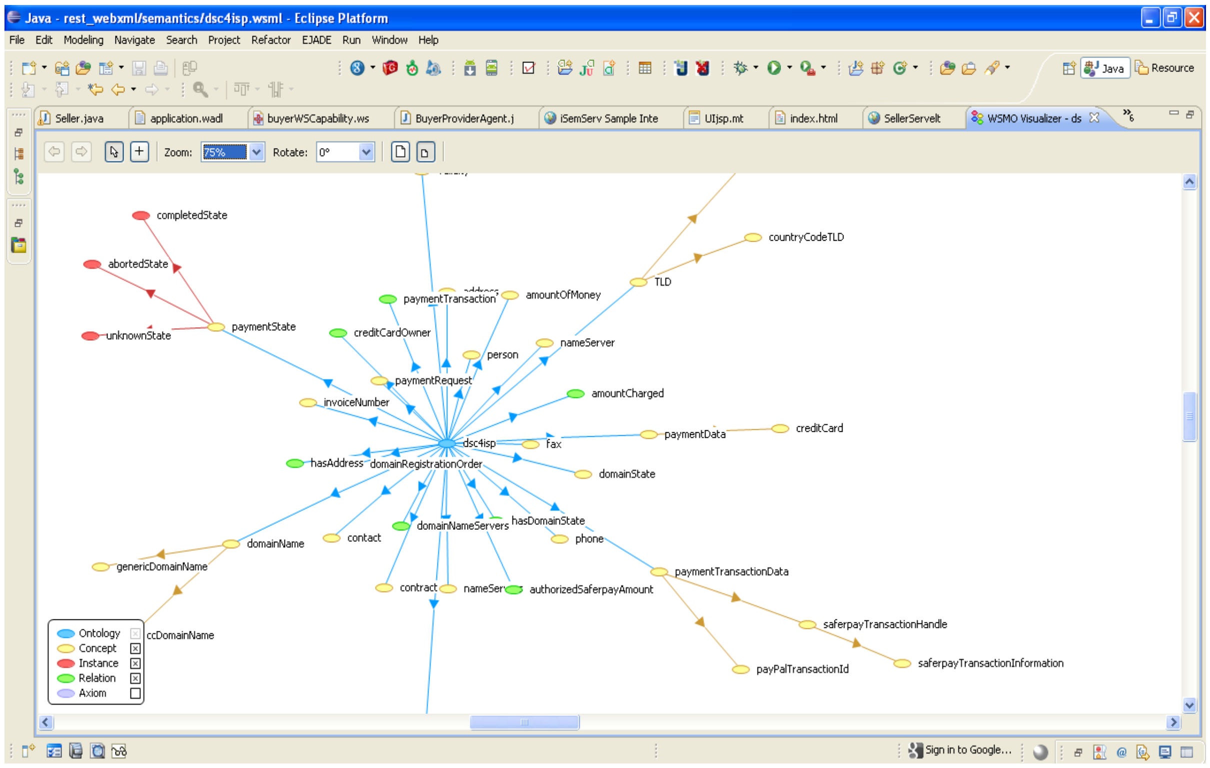


Figure 8.13: Imported Amazon Ontology Visualization

The Amazon ontology was simply imported into the project by using the WSMO editor launcher. The launcher could also be used to edit all the various elements of the domain ontologies or the semantic descriptions.

As may be noted from the illustrated online multimedia trading scenario, the process of engineering intelligent semantic services is extensive; and service developers need methods and tools to ease and speed up such a process. From the tested scenario above, we have demonstrated how this could be realized using the *i*SemServ framework.

In the following section, a discussion is provided with regard to the evaluation results extracted from the presented scenario, and others that have been partially implemented, using the *i*SemServ proposed solution.

8.3. SCENARIO EVALUATION DISCUSSIONS

The evaluation approach of using the proposed framework to partially implement use case scenarios has provided several insights with regard to the engineering of intelligent semantic services. The following benefits, in terms of the proposed solution, were observed:

- **Uniformity:** The engineering of different building blocks that make up intelligent semantic services can be realized within a unified environment.
- **Acceleration:** Any development effort is reduced through the auto-generation of different implementation artefacts, which may lead to high development times and costs, if a manual approach is chosen.
- **Control:** The service engineer controls the engineering life cycle, and the *i*SemServ plug-in does impose restrictions on the types of service or the semantic descriptions. The only requirement pertains to the usage of UML-based service models for structuring services and domain knowledge.
- **Simplification:** The service engineer need not to be concerned with the generics, but rather need to focus on the specific implementations for intended services. The generation of semantic descriptions and domain ontologies is made understandable through the mappings with service models. The addition of intelligence properties in semantic services is simplified, and the testing of such services can be further simplified, through the auto-generation of simple user interfaces, which could save the developer time and effort in testing implemented intelligent semantic services.
- **Interoperability:** The external and internal tools interoperate effectively with the *i*SemServ platform to simplify the generation of domain ontologies and semantic descriptions, and for the visualization of ontologies– thereby minimizing the steep learning curve of semantic technologies and models.

- **Integration:** The existing Web service tools, such as a WSDL generator, can easily be integrated with semantic technologies, such as WSMO, to form semantic services.
- **Domain-independency**: The evaluation also demonstrated that the proposed *i*SemServ solution is not dependent on a specific domain, as intelligent semantic services could be engineered for multiple domains (e.g. online trading).
- **Elementary Intelligence**: The wrapping of semantic services with intelligence follows standard approaches, such as Object-Oriented Design, where wrapping is achieved at a message-level and a knowledge-level, using common Java classes and agent development environments, such as JADE.

The *i*SemServ framework presented in this study is research-oriented, and as a result, has some practical limitations. For instance, key features, such as dynamic semantic services discovery, selection, composition, and monitoring are not addressed. Nevertheless, one of the key principles of our solution is *extensibility*, which is intended to enable other researchers and developers to extend the *i*SemServ platform with any required modules via the Eclipse environment.

The following section presents additional evaluation results that were derived by way of a comparative analysis. This analysis focused on comparing the *i*SemServ framework with existing solutions that have objectives closely aligned with the goals of this study.

8.4. COMPARATIVE ANALYSIS

Currently, there are no commercial platforms available for facilitating the process of building intelligent semantic services. However, research that has been done in this field over recent years suggests that the next generation platforms for developing software systems will focus on semantics-enabled systems.

The existing solutions that formed part of the analysis were discussed and summarised in Chapter 2 (cf. Section 2.5).

8.4.1. Comparison Criteria

The comparative criteria used for evaluating the suggested solution against similar solutions in literature are based on the design principles presented in Chapter 6 (cf. Section 6.2). Figure 8.14 depicts an overview of the design requirements. The requirements are divided into *simplification*, *acceleration*, and *intelligence*. With regard to *simplification*, the proposed framework is compared with the existing solutions according to the following criteria:

- *Model-driven:* Model-driven engineering approaches are meant to enable software developers to increase productivity and shorten the software development life cycle. Henceforth, the premise in this study is that any solution that attempts to simplify and accelerate the process of engineering semantic services needs to follow a model-driven approach.
- *Decoupling:* Syntactic services, semantics, and intelligence-building blocks need to exist independently of each other; but they still need to be able to interoperate.
- *Complexity hiding:* semantic descriptions, domain ontologies, and intelligence complexities need, to some extent, to be hidden from service developers.
- *Interoperability:* The solution needs to enable different tools to interoperate. For instance, syntactic Web services tools should easily interoperate with other semantic Web services tools.
- *Visualization:* Large domain ontologies can be complex to understand without the assistance of the correct tools. Thus, any solution suggested for simplifying and accelerating the process of engineering intelligent semantic services needs to provide some methods and tools for visualizing complex domain ontologies and semantic descriptions.

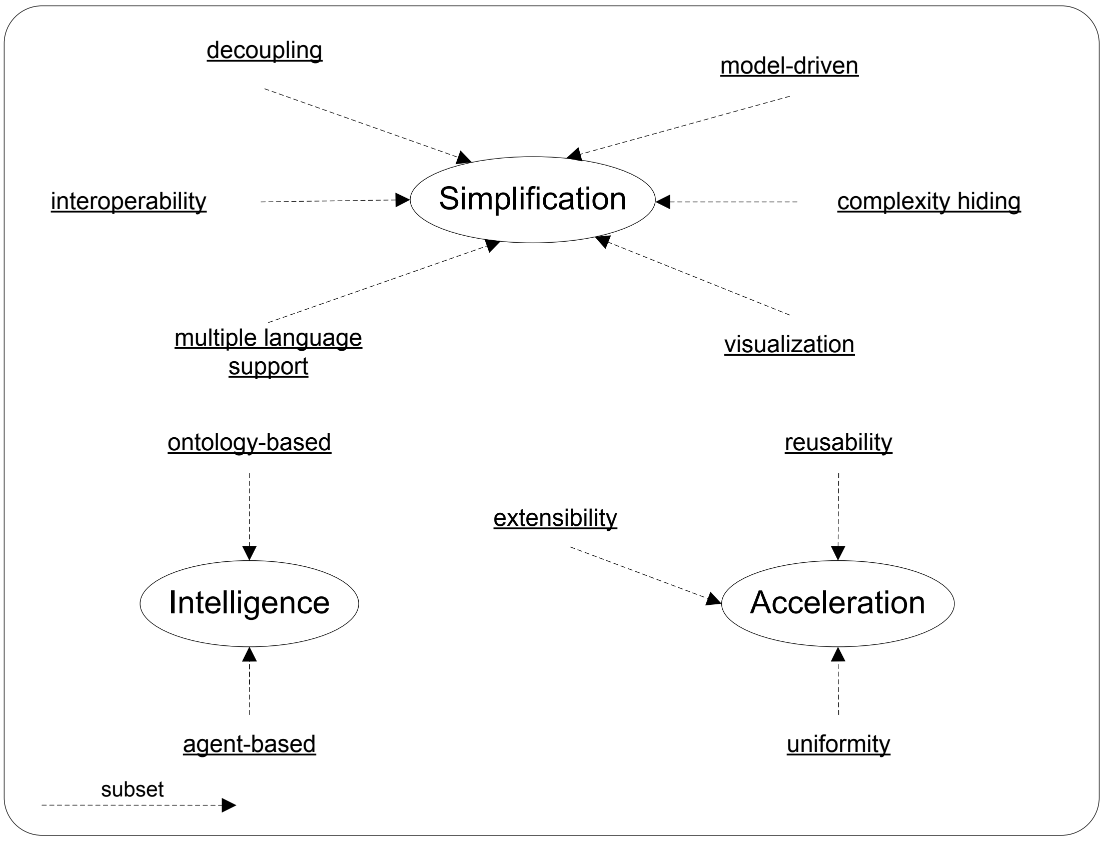


Figure 8.14: *i*SemServ Design Principles

- *Multiple Language Support:* Any solution that attempts to solve the challenges of engineering semantic services needs to support multiple semantic models, such as WSMO or OWL-S.

With regard to *acceleration*, the proposed framework is compared with existing solutions according to the following criteria:

- *Re-usability:* Any solution that seeks to address the challenges of engineering semantic services should re-use the generic and mature Web services technologies. In addition, any semantic solution should enable the re-use of the existing domain knowledge and semantic descriptions.
- *Extensibility:* Any solution that attempts to address the challenges raised in Chapter 1 (cf. Section 1.2) should be extensible without extensive modifications, to be able to accommodate the integration of additional

modules, such as service composition, multiple service standards and semantic description languages.

- *Uniformity:* Any solution that supports the building of semantic services needs to do so within a uniform environment, where service engineers are not expected to switch between fragmented tools to deliver functionally intelligent semantic services.

In terms of the *intelligence* principle*,* the proposed framework is compared with the existing solutions according to the following criteria:

- *Ontology-based:* Any solution that addresses the challenges of building semantic services needs to consider different standards of representing domain ontologies and semantic descriptions.
- *Agent-based:* Semantic services and software agents are viewed as complementary (Garcia-Sanchez *et al.*, 2009). Thus, it is maintained that novel technologies attempting to address the challenges of engineering semantic services need to consider the mapping of agents and semantic services, in order to achieve automation in service discovery, selection, composition, and execution.

8.4.2. Qualitative Comparison

Table 8.1: *i*SemServ Comparative Analysis

Principles / Solution	Model-driven	Complexity Hiding	Decoupling	Interoperability	Multiple Language Support	*V*isualization	Reusability	Extensibility	Uniformity	Ontology-based	Agent-based
	Simplification						**Acceleration**			**Intelligence**	
iSEMSERV	√	√*	√	√	√*	√*	√	√	√	√	√
SEMMAS	×	×	√	√	√	×	√	×	×	√	√
SWF	×	√*	×	√	×	√*	√*	×	×	√	√
OWL-S IDE	×	√	√	√	×	√*	√	√	×	√	×
WSMO Studio	×	√	√	√	×	√	√	√	×	√	×
INFRAWEBS	√*	√	√	√	×	√	√	√	√*	√	×
ODE-SWS	√*	√	√	√	√*	√*	√*	×	×	√	×
IRS-III	×	√	√	√	√*	√	√*	√	×	√	×
LEGENDS	? = not known × = not addressed	√ = addressed √*= partially addressed									

8.4.3. Qualitative Comparison

In Table 8.1, a summary of the comparative analysis is presented. The *i*SemServ solution adopts a *model-driven* approach, due to its objective to increase software development productivity and efficiency. Furthermore, the model-driven approach is used across all layers – from the services layer to the intelligence layer. INFRAWEBS (Agre *et al.*, 2007) and ODE-SWS (Corcho *et al.*, 2003) are the only solutions in the comparative analysis that also adopt the model-driven approach. However, as indicated in Table 8.1, these solutions partially support the model-driven technique in the engineering life cycle of semantic services.

For instance, the service developer can only model WSMO semantic features using the INFRAWEBS environment. The modelling of services (e.g. SOAP or REST) is, however, not supported. The INFRAWEBS graphical modelling module is tightly coupled to the solution itself, and is non-conformant to the model-driven architecture (MDA). The ODE-SWS solution focuses mainly on the semantic descriptions that could be implemented using different semantic languages. However, the modelling part is only supported by an internal ODE-SWS graphical tool; which also does not accommodate the modelling of other artefacts, such as syntactic services and intelligence.

The majority of the solutions evaluated pay attention to the principle of *complexity hiding* – when it comes to simplifying the process of engineering semantic services. Our solution addresses the issue of complexity hiding through the auto-generation of skeleton code. However, the developer still needs to understand the different languages, in order to augment the generated semantic descriptions and ontologies. The only solution that does not address the issue of complexity hiding is the SEMMAS solution (Garcia-Sanchez *et al.*, 2009).

In SEMMAS, the developer is required to manually generate all the necessary building blocks (e.g. ontologies) that comprise an intelligent semantic service. On the contrary, Semantic Web Fred (SWF) (Stollberg *et al.*, 2004) partially addresses the principle of complexity hiding by generating proprietary ontologies from XML Schema Definition (XSD) files, using an ontology management unit called Ontology Tower.

With regard to the *decoupling* principle, the only solution found to be lacking is SWF. In this regard, SWF couples services, and ontologies into FREDs, which are basically software agents.

*A*ll of the evaluated solutions address the principle of *interoperability*, although, the principle is addressed in various ways. For instance, in OWL-S IDE (Srinivasan, Paolucci & Sycara, 2006), syntactic service technologies (e.g. UDDI and WSDL standards) are interoperated effectively with specific semantic technologies (e.g. OWL Editor). In SWF, the *interoperability* principle in the context of this study is limited to SOAP and compiled ontologies' interoperability.

In terms of the *i*SemServ solution, the *interoperability* requirement is addressed across different layers, where multiple syntactic services technologies are easily interoperated with semantic technologies and agent-based technologies. SEMMAS also address the *interoperability* principle in a similar way to the iSemServ framework.

In terms of supporting the multiple standards and languages, the evaluation results revealed that only *i*SemServ and SEMMAS consider this requirement across the services and semantics layers. For example, in *i*SemServ both RESTful and SOAP services are accommodated in the service layers, and heavy-weight and lean semantic description approaches are also supported through the application of UML profiles. SEMMAS does not put any restrictions on ontology languages that can be used to semantically describe services. With regard to the services layer, RESTful and SOAP services are both supported by the SEMMAS framework.

ODE-SWS (Corcho *et al.*, 2003) uses WebODE (WebODE, 2003), an ontology engineering workbench that enables the use of different ontology standards in the semantics layers. The other solutions, such as IRS-III (Domingue *et al.*, 2008), mainly support *multiple languages* by integrating their own ontology standard (OCML) and WSMO within the semantics layer.

The *visualization* requirement suggested for simplifying the viewing of semantic descriptions and domain ontologies by service engineers is supported either partially or completely by the majority of the solutions selected for the evaluation. The only solution that does not address the *visualization* of complex ontologies or semantic descriptions is SEMMAS. It merely provides different user interfaces that the service developer could use to point to where ontologies and semantic descriptions are located.

The comparative analysis task also revealed support for the *re-usability* principle by all solutions evaluated. INFRAWEBS, WSMO Studio, OWL-S IDE, SEMMAS, and iSemServ fully support the re-use of existing Web services technologies and existing domain ontologies or semantic descriptions. Other solutions, such as SWF, IRS-III, and ODE-SWS focus exclusively on the re-use of services and technologies, but not necessarily on domain knowledge or semantic descriptions.

The *uniformity* principle basically contributes to the acceleration of the engineering process. From the evaluations, it was found that only the *i*SemServ solution fully facilitates the engineering of semantic services within a unified environment. INFRAWEBS partially addresses the uniformity principle by facilitating the import of existing syntactic services and the creation of semantic descriptions for imported services. However, in INFRAWEBS, the process of building intelligence modules for the formation of intelligent semantic services is not considered.

The other solutions focus exclusively on the engineering of semantic descriptions and/or ontologies, and leave the service engineer to develop syntactic services, and intelligence features using external solutions.

In terms of *intelligence* wrapping over semantic services, all the solutions address the *intelligence* principle from the perspective of ontologies. Thus, all the solutions that were part of the evaluation process are *ontology-based*. However, the evaluation process further compared the solutions in terms of *agent-based intelligence*, which is considered paramount in the processing of semantic descriptions and domain ontologies with minimal human intervention. Only three solutions, that is, *i*SemServ, SEMMAS, and SWF – clearly address *agent-based* *intelligence,* as one of the requirements identified for any solution that addresses the issue of simplifying and accelerating the process of engineering intelligent semantic services.

In Table 8.2, the main features of the *i*SemServ framework are further compared with those of similar solutions. The distinction was evaluated, based on the conformity to the *agent-based* principle, which was further classified into three features: (1) *Message-level:* semantic services and software agents communicate via messages, using syntactic services protocols, such as SOAP and RESTful-HTTP, (2) *knowledge-level:* semantic services and agents communicate using shared ontologies, and (3) *rules and reasoning*: these agents are enabled to reason on semantic service rules and ontologies.

In addition, the support of *complexity hiding* in different solutions was further analyzed, based on code-generations, which is viewed as one of the key approaches for simplifying and accelerating the process of building intelligent semantic services in this study.

From the evaluation, it became clear that the solution addressing both *message-level* and *knowledge-level* integration of semantic services and software agents is *i*SemServ, whilst SEMMAS and SWF solutions address the integration, particularly at the knowledge-level, and by using domain ontologies.

Table 8.2: *i*SemServ Core Features

Features / Principles	Message-level	Knowledge-level	Rules and Reasoner	Code Generations
	Agent-based			Complexity-Hiding
iSEMSERV	√	√	√*	√*
SEMMAS	×	√	√*	×
SWF	×	√	√	×
OWL-S IDE	×	×	?	√*
WSMO Studio	×	×	√	×
INFRAWEBS	×	×	√	√*
ODE-SWS	×	×	√	√*
IRS-III	×	×	√	×
LEGENDS	? = not known × = not addressed √*= partially addressed √ = addressed			

The analysis shows that all of the solutions, except OWL-S IDE fully or partially support the definition of rules and reasoning over domain ontologies. However, different techniques are used across the evaluated solutions. In our proposed solution, partial JESS and ontological rules are auto-generated from rules-annotated service models. For reasoning purposes, the JESS inference engine (Friedman-Hill, 2003) is exploited, as was explained in the previous chapter.

In INFRAWEBS, Prolog engines are used for storing rules, and performing matching tasks during service discovery and selection. SEMMAS only touches on the mapping rules, and the reasoning features are supported by hard-coded software agents.

IRS-III supports rules and reasoning features through chaining rules and the OCML reasoner (Domingue *et al.*, 2008) for processing and matching semantic descriptions to relevant semantic services. On the contrary, the ODE-SWS solution relies on the WebODE workbench for reasoning over ontologies. The rules that are considered for reasoning purposes are mainly included in the ontology. As with ODE-SWS, the SWF framework facilitates the definition of rules using the selected ontology language.

Finally, the WSMO studio uses state transition rules (Dimitrov *et al.*, 2007) and exploits multiple WSML reasoners for processing domain ontologies and semantic descriptions.

In terms of code-generations for complexity hiding, *i*SemServ is capable of facilitating auto-transformations from the service layer to the intelligence layer. Other solutions, such as OWL-S IDE, INFRAWEBS, and ODE-SWS, were found to be supporting code-generation mainly for semantic descriptions, but not for syntactic services and intelligence implementation.

The comparative analysis presented in this section revealed some distinctive differences between our proposed solution and similar existing solutions. Where essential features were found to be supported across other solutions, the differences between those features also revealed how *i*SemServ addresses the specific challenges of building intelligent semantic services.

The next section will present the final evaluation activity with regard to the performance and scalability of our proposed solution.

8.5. SCALABILITY AND PERFORMANCE

In this section, an additional evaluation activity that was conducted is presented and discussed. This specific evaluation was conducted using the SEALS methodology for evaluating semantic technologies. It should be noted that none of the related solutions that formed part of the comparative analysis were evaluated using the SEALS methodology. Thus, this work goes one step further by considering the scalability and performance of the *i*SemServ platform with regard to the simplification and acceleration of the process for engineering intelligent semantic services.

For the purposes of this study, we only focused on two evaluation principles, *performance and scalability*. Other evaluation criteria considered by SEALS as highlighted in Chapter 1 (cf. Section 1.6), such as *solution correctness* were not considered, as they are beyond the scope of the *i*SemServ framework. However, *usability* evaluations were not conducted due to the evident simplicity of the *i*SemServ plug-in as demonstrated in Section 8.2 (cf. Figure 8.4).

As discussed in Chapter 1 (cf. Section 1.6), *scalability* refers to the capability of the evaluated solution to perform tasks involving an increasing number of service descriptions, ontologies, and semantic descriptions. *Performance* refers to the functioning of specific semantic service activities, such as service implementation, deployment, discovery, and execution. In general, the SEALS methodology suggests that the performance of identified activities is determined by using the execution time and the throughput.

In evaluating *i*SemServ performance and scalability conformity, a series of experiments was conducted. The experimental environment was set up on a Microsoft Windows XP (32-bit OS), Intel Core™ 2 Duo CPU (2GHz), and a 1GB RAM Acer Travel-Mate 6492 machine. Eclipse memory was capped at 512MB. In measuring the approximate execution times of the different modules of *i*SemServ, an Acceleo Profiler embedded in Eclipse was used.

The Acceleo Profiler provides features for identifying and isolating performance problems, such as resource limitations and bottlenecks. Additionally, it covers performance monitoring, execution, tracing and profiling, and logging. It was chosen for conducting different performance and scalability tests, mainly because the proposed solution exploits code generation techniques and a template engine provided by the Acceleo platform, as was discussed in Chapter 7.

Using real-life project scenarios, a number of service models were manually designed using the Eclipse UML SDK. The models comprised 6 to 1152 classes, annotated with **<<RESTful>>** and/or **<<WSMO>>** keywords. In Figure 8.15, a graph is depicted highlighting the performance of the proposed *i*SemServ platform under varying service model sizes. The overall execution times, presented in seconds (s) illustrate the time it took the platform to generate the artefacts involved in engineering intelligent semantic services. The objective was not about showing accuracy in terms of execution times, but to relatively demonstrate the performance and scalability of the proposed framework under varying requirements.

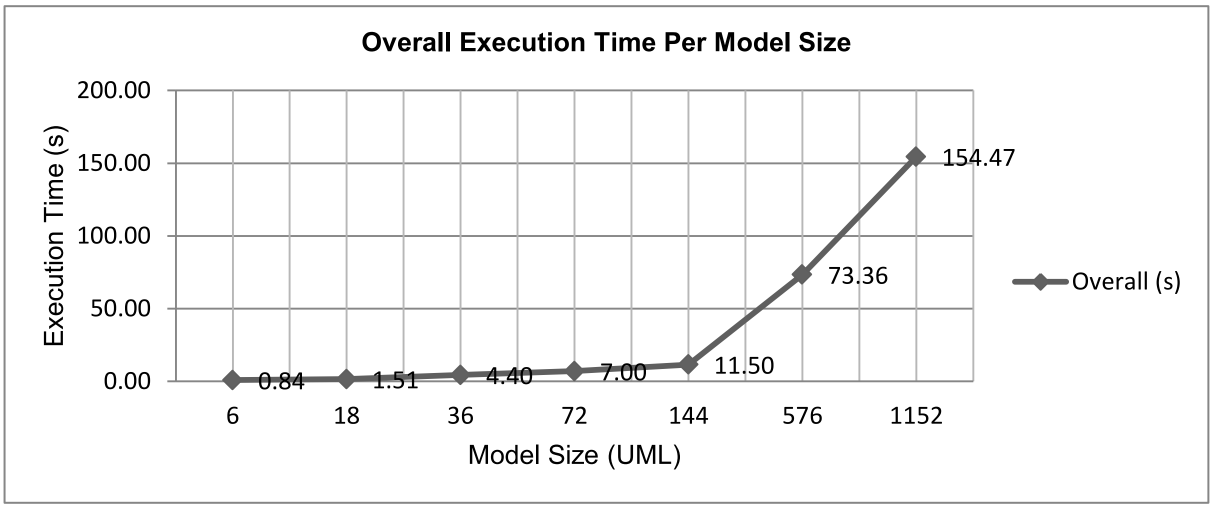


Figure 8.15: Overall Performance of iSemServ Platform

Figure 8.15 suggests that, as the size of the service model grows, so does the amount of time required to automatically transform the model to different artefacts. For instance, processing a service model with 144 classes required about 11 seconds on average. An average of 73 seconds was additionally required to generate a total number of 576 Java classes and mapped building blocks, such as semantic descriptions. The code generation execution time increased to almost 3 minutes on average for processing a service model made up of 1152 classes.

In Table 8.3 and Figure 8.16, average execution times, based on 10 experimental runs in relation to code generated for different building blocks, are illustrated.

Table 8.3: Experimental Data (Averages)

Size (ms)	Models (ms)	RESTful Services (ms)	Syntactic Descriptions (ms)	Domain Ontologies (ms)	Semantic Descriptions (ms)	Agents and JESS Rules (ms)	Test-User Interfaces (ms)	Deployment Descriptors (ms)	Internal (ms)	Total Time(ms)	Total Time(s)	Total Time (m)
6	2.50	97.10	78.15	94.50	78.50	101.80	63.50	281.10	46.40	843.55	0.84	0.01
18	12.50	456.10	114.30	140.10	125.50	110.20	78.50	406.50	62.80	1506.50	1.51	0.03
36	20.90	1000.60	532.20	844.00	875.00	481.40	78.10	532.70	31.20	4396.10	4.40	0.07
72	18.60	1560.90	951.10	999.30	841.00	1201.00	380.40	1036.10	16.50	7004.90	7.00	0.12
144	48.10	2109.40	1625.40	750.10	1287.90	865.10	594.80	4203.40	16.10	11500.30	11.50	0.19
576	90.50	14303.10	12867.50	10918.60	11503.90	12744.10	1565.40	9308.50	62.50	73364.10	73.36	1.22
1152	245.70	27241.80	26175.40	19873.70	30357.90	21876.40	10670.60	17801.20	230.00	154472.70	154.47	2.57

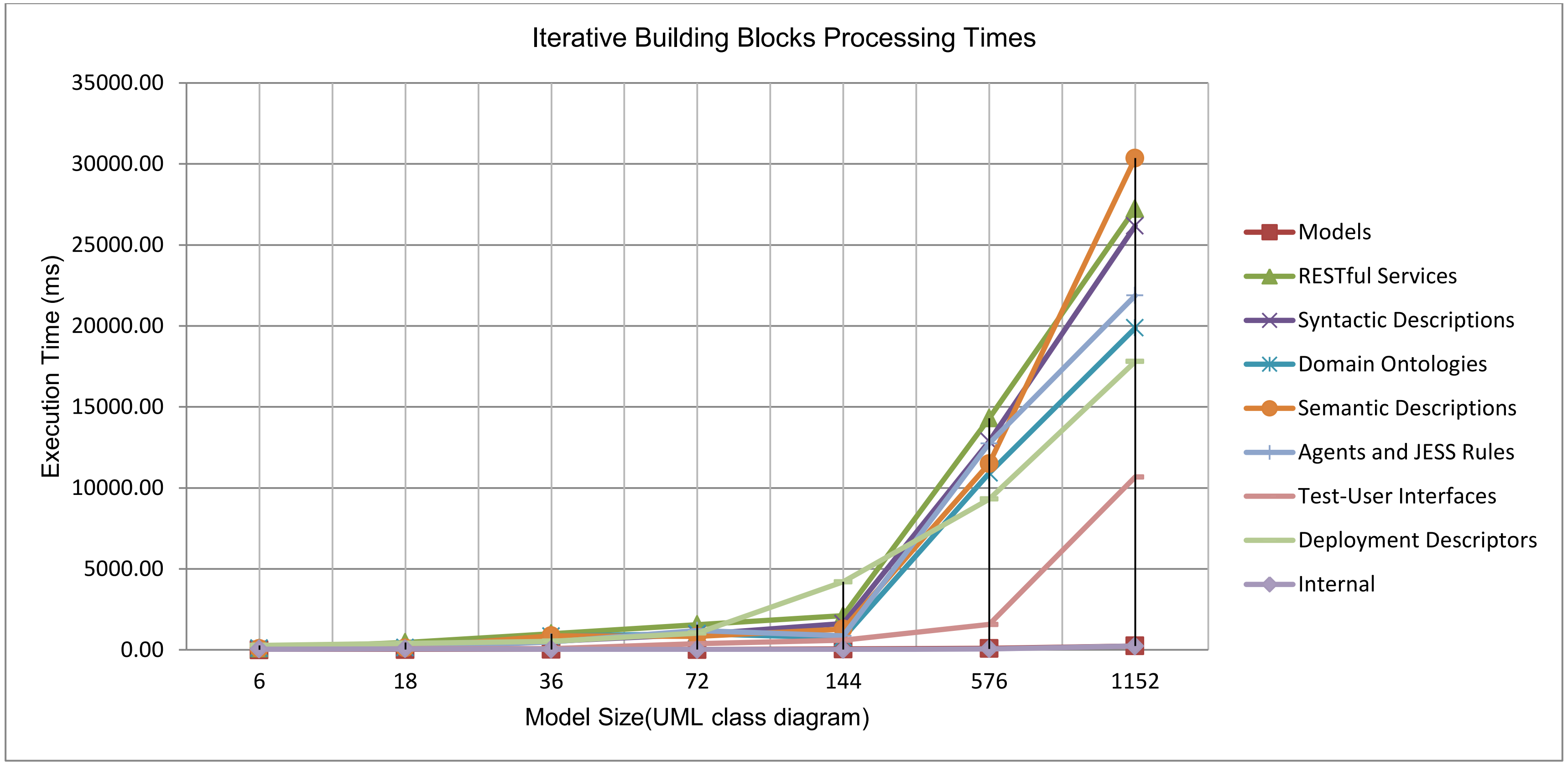


Figure 8.16: Iterative Building Blocks Processing Times (Average)

Table 8.3 denotes model sizes, number of RESTful services, domain ontologies, semantic descriptions as well as all the other essential building blocks generated during the experiments.

Overall, the auto-generation of semantic descriptions in the semantics layer took most of the processing time (cf. Figure 8.16) as compared with other artefacts in the services and intelligence layer, as the service model surged. The processing of the service models required fairly a small amount of time (e.g. 48 milliseconds for a service model made up of 144 classes). As the service model size grew to 1000+ classes, the processing time increased slightly to 245 milliseconds (ms). Withal, this is still viewed as manageable, and suggests that an increasing model size would not introduce major processing challenges.

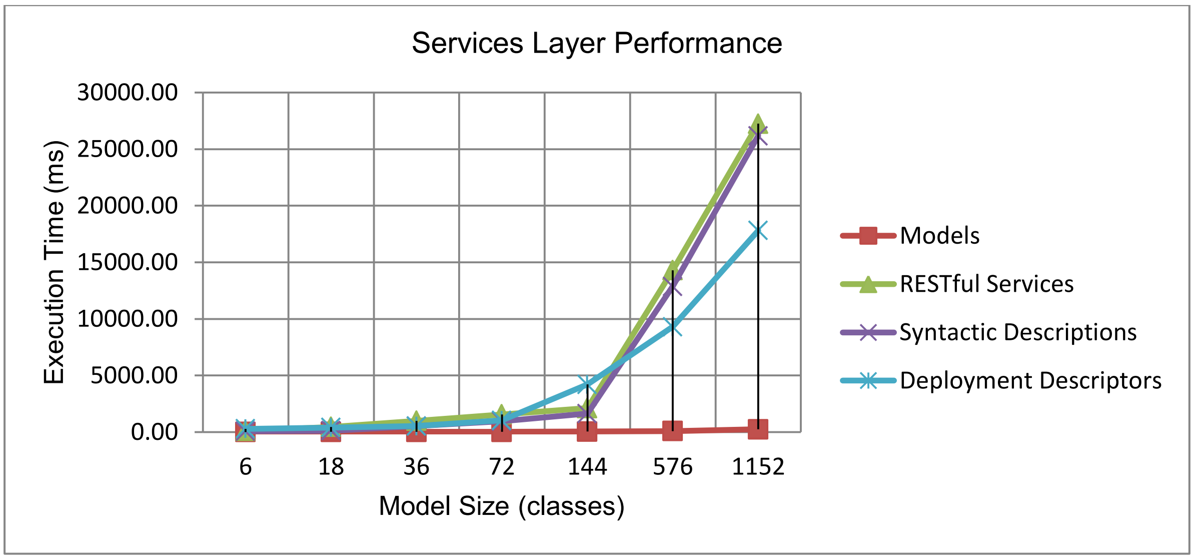


Figure 8.17: Services Layer Processing Times

The processing time required for auto-generating syntactic service skeleton code, syntactic descriptions, and deployment descriptors increases with the model size (cf. Figure 8.17).

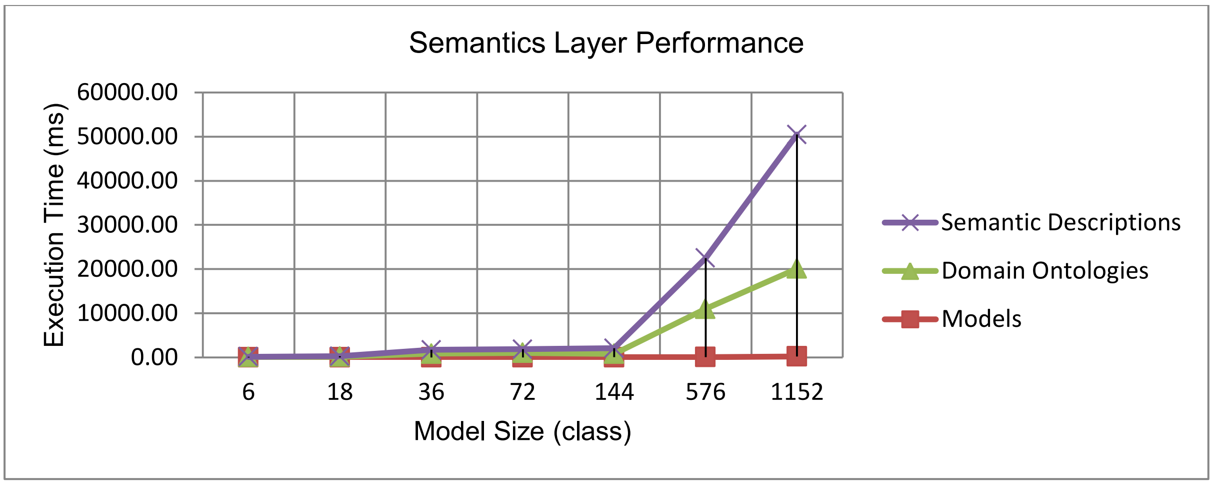


Figure 8.18: Semantics Layer Processing Times

With regard to the semantics layer (cf. Figure 8.18), the processing time required to generate semantic descriptions increased from 11 to 30 seconds when the service model size double from 500+ to 1000+ classes. However, the processing time required to generate domain ontologies from varying service models remained minimal; that is, from 500+ to 1000+ classes an additional 9 seconds was needed.

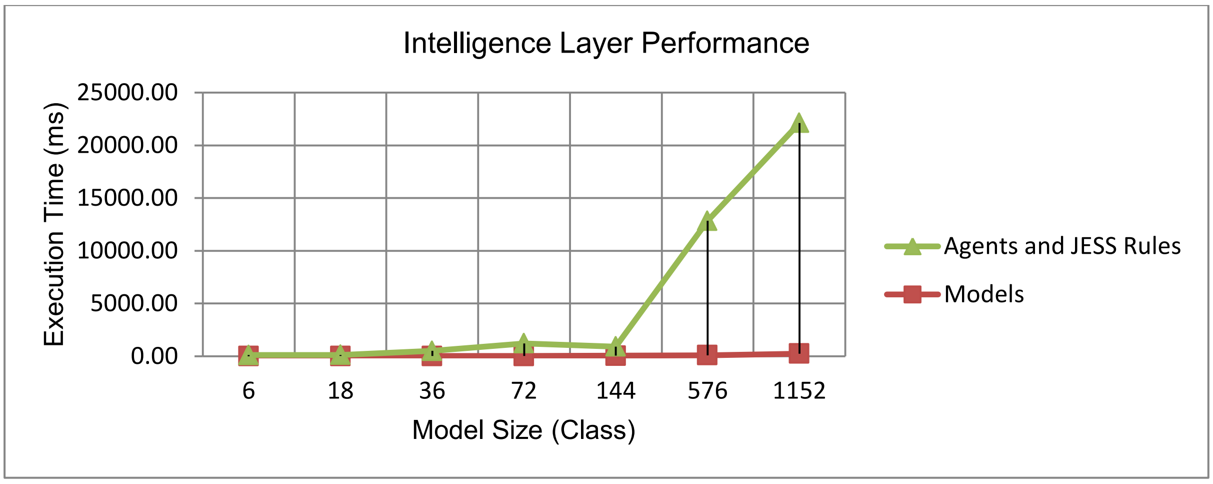


Figure 8.19: Intelligence Layer Processing Times

In Figure 8.19, the processing times required to transform the varying service models to the intelligence artefacts (i.e. agents' code and rules) are illustrated. It required an execution time of less than 5 seconds on average to generate all the necessary intelligence codes, when the model size ranged between 5+ and 100+, which for many small software projects is quite sufficient. As the model size increased to between 500+ and 1000+ classes, the processing time requirements also increased to around20 seconds – to generate all the intelligence artefacts. As may be noted in Figure 8.19, a slight drop in processing time between model size (144) and model size (72) is noticeable. The drop could be attributed to a number of reasons, such as the number of processes in the computer memory or the number of activities in Eclipse during specific experiment runs.

Other modules that formed part of the experiments, although not essential to the proposed solution, included the auto-generation of web-based user interfaces for purposes of simply and quickly testing the generated building blocks. The processing time required to generate different user interfaces for a 1000+ classes' service model was approximately 10 seconds.

From the experiment, it may be concluded that the majority of the processing time is taken by the services and the semantics layer. The services layer (e.g. RESTful services, syntactic descriptions, and deployment descriptors) consumes the major portion of the processing time. In order to ensure that the platform is not burdened with a lot of processing time, the platform has also been designed in a manner that enables the service engineer to select specific building blocks to generate at a time. Thus, it is not a requirement of the system to auto-generate all the artefacts within a single iteration.

The transformation process can be split into different phases, thereby minimizing the processing load and improving the performance of the platform.

In addition, it may also be concluded that the *i*SemServ platform is scalable, in the sense that it is capable of handling an increasing number of classes in the service model without any major challenges except for a small number of "out-of-memory" exceptions in Eclipse, which could be sorted out by increasing the maximum memory for Eclipse. Furthermore, our model is inherently scalable on account of its style of implementation (i.e. plug-in) in Eclipse.

The scalability and performance analysis further indicates that the *i*SemServ platform is capable of simplifying and accelerating the process of engineering intelligent semantic services. This is demonstrated by the fact that the amount of processing time required to generate the skeleton code for different layers is smaller compared with the time that the service engineer would need to manually generate all the different artefacts.

Our analysis also demonstrated that as the model size increases, so does the required processing time. However, the processing time was still immeasurably small when compared with that required by the manual process.

8.6. SUMMARY

This chapter has presented the evidence in relation to the practicality, relevance, novelty, performance, and scalability of the proposed solution. The first evaluation focused on the practicality and relevance of our solution within an online multimedia trading domain, where semantic services could be of use. A scenario was defined and a service model developed. The model was then fed into our platform to demonstrate how it could simplify and accelerate the process of engineering intelligent semantic services.

Using the implemented *i*SemServ Eclipse Plug-in, it was demonstrated how the different building blocks could be generated.

The qualitative evaluation, in terms of a comparative analysis, was also presented – demonstrating thereby the main differences between the proposed solution and the existing solutions. From the comparative analysis, it was revealed that our proposed solution introduced the approach of firstly building intelligent semantic services within a unified environment. Secondly, our solution adopts a model-driven approach, where all the necessary modules required to engineer intelligent semantic services are derived from the service model annotated with defined UML stereotypes.

In addition, software agents and associative JESS-rules are automatically generated to wrap semantic services via knowledge and message levels for the purpose of automatically processing services and reasoning over domain ontologies and semantic descriptions. Lastly, our solution addresses the issue of complexities in building intelligent semantic services by proposing a template-based approach for code generations and the support of different architectural styles and semantic description languages.

Finally, the results, from a series of experiments, demonstrated that our solution is capable of handling an increasing size of service models; this includes an increasing size of services, service descriptions, domain ontologies, semantic descriptions, and intelligent agents. This approach was not followed by the similar solutions that the proposed iSemServ solution was compared with.

In the following chapter, a summary and conclusion of the study is provided.

CHAPTER 9: Summary, Conclusion, and Further Research

In this chapter, we summarize the thesis by reviewing the research problem, the research questions, and the extent to which they were addressed. Furthermore, we highlight the main contributions (theoretical and practical) derived from this study. The remaining challenges and limitations of the study are also discussed. Further research work that could address some of the identified limitations and challenges is also discussed.

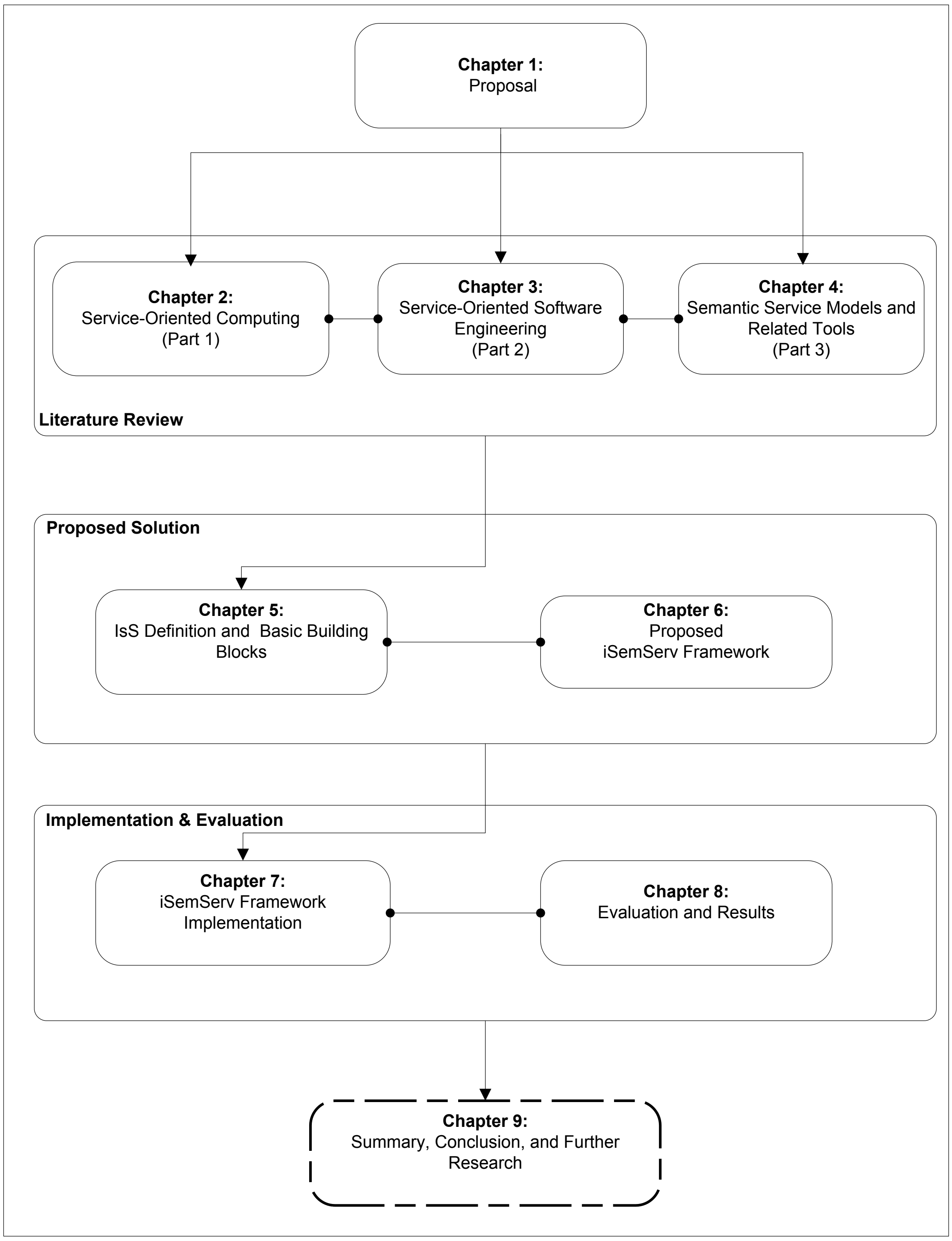

Chapter 1:
Proposal
Chapter 2:
Service-Oriented Computing
(Part 1)
Chapter 3:
Service-Oriented Software Engineering
(Part 2)
Chapter 4:
Semantic Service Models and Related Tools
(Part 3)
Literature Review
Proposed Solution
Chapter 5:
IsS Definition and Basic Building Blocks
Chapter 6:
Proposed iSemServ Framework
Implementation & Evaluation
Chapter 7:
iSemServ Framework Implementation
Chapter 8:
Evaluation and Results
Chapter 9:
Summary, Conclusion, and Further Research

Figure 9.1: Overall Thesis Structure

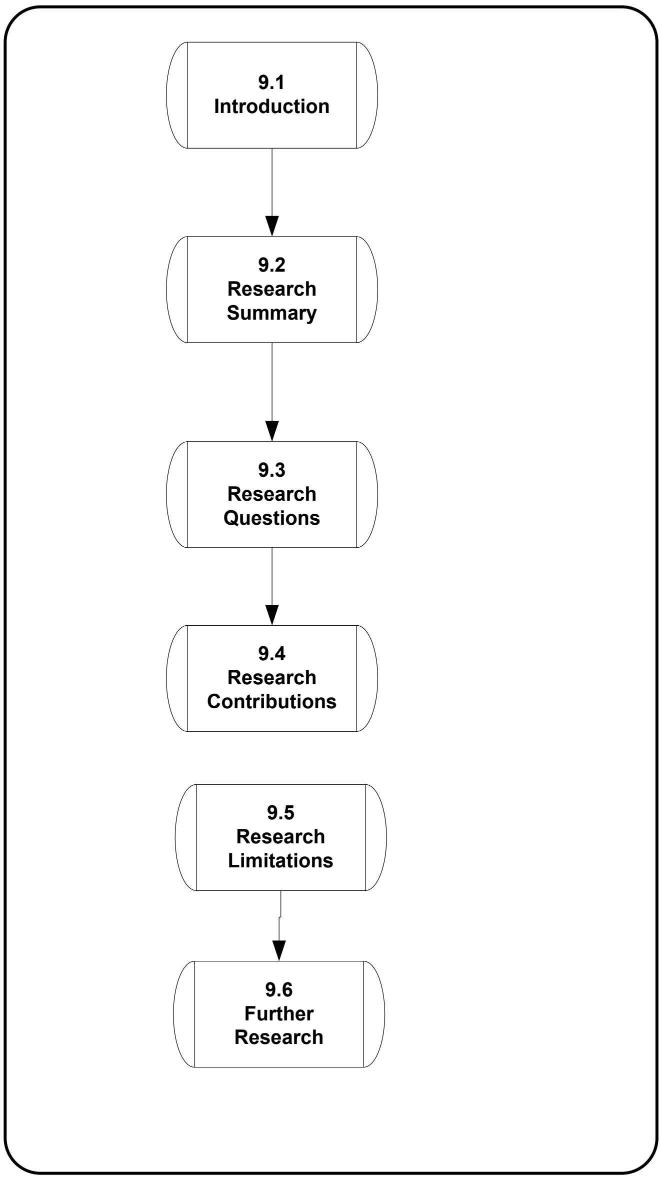


Figure 9.2: Chapter 9 Layout

9.1. INTRODUCTION

In this final chapter, we summarize our work by revisiting the research problem that inspired this study. We then review the extent to which the identified research questions were addressed. In addition, the main contributions emanating from the proposed solution are highlighted, from a practical and theoretical point of view.

Furthermore, the research limitations and challenges identified are highlighted; and further research work that could address some of these identified challenges is discussed.

9.2. RESEARCH SUMMARY

Semantic services are touted as the next generation of the future Web, where business processes would be executed and automated by machines with minimal user interventions. However, the implementation and development of such services have been lacking in real-life environments. The lack of implementation and development of these services is attributed to a number of challenges, such as tedious and error-prone semantic service development processes, the lack of integration of semantic technologies with expansive Web service technologies, the steep learning curves of semantic description languages, the lack of unified platforms that support the simplification of the process of engineering semantic services (Siorpaes & Simperl, 2010), and the lack of semantic platforms that provide end-to-end development of intelligent semantic services.

In addressing some of these challenges, we exploited a number of techniques, guided by the following summary of the main research question and the supporting questions.

9.3. RESEARCH QUESTIONS

In this section, we review how the supporting questions were addressed – in answering the main research question and accomplishing the main objectives of this thesis. The main research question was phrased as follows: *How could a unified service creation framework simplify and accelerate the process of engineering* *intelligent semantic services (IsS)?* A review of the supporting questions and essential objectives that have been accomplished in this thesis is as follows:

SQ[52]1: *What are the fundamental building blocks that constitute an intelligent semantic service and the characteristics thereof?*

In coming up with an approach that aims to deal with the challenges of building intelligent semantic services, it was of importance for us to understand a number of issues, such as: *What is an intelligent semantic service? How is it different from existing services concepts, such as Web services? What components comprise such services? What are the characteristics of such components?*

In tackling these supporting questions, a literature review, the thesis problem space, and the identified research objectives constituted our valuable compass.

The supporting question (i.e. SQ1) was covered by providing an elaborative definition of the term *intelligent semantic service* (*IsS*). This was motivated by the fact that at the time of this study, there was no common definition of the term. The proposed definition is provided in Chapter 5 (Section 5.2). This was then followed by the identification of the fundamental building blocks that could compose a functional intelligent semantic service.

The key building blocks that were identified were grounded in the concepts of Web Services, Domain Ontologies, Semantic Web Services, and Intelligent Agents.

Furthermore, the main characteristics of the building blocks were formulated and presented in Chapter 5.

SQ2: *How could service engineers develop intelligent semantic services from the identified fundamental building blocks?*

[52]**SQ:** supporting question

Once the fundamental building blocks were devised, it was then important to understand the manner in which these components could be utilised by service engineers – to build functional intelligent semantic services. A model-driven engineering methodology was proposed to address this supporting question. It was proposed because of its benefits and relevance to our problem space, such as enabling reduced development times for new services, and open integration of semantic technologies with expansive technologies (e.g. Web services). The proposed MDE methodology consists of six steps (cf. Chapter 6, Section 6.3), which provide a stable foundation for simplifying the process of engineering intelligent semantic services.

SQ3: *What are the requirements for designing and developing a unified service creation framework, in order to simplify and speedup the process of engineering IsS?*

The main objective of the study was to investigate and formulate a unified service creation framework, so as to simplify the complexities involved in engineering intelligent semantic services. In an attempt to design and develop such a solution, it became essential to specify the design requirements. These requirements, which are presented and discussed in Chapter 6 (Section 6.2) emanated from the objectives of the study, the literature review, related work, and the components that make up an intelligent semantic service, as derived from one of the supporting questions (i.e. SQ1).

The design requirements provided basis for designing and developing a solution that would not only address the challenges of building semantic services, but would further ensure that our solution is future-proof in terms of extensibility and scalability, re-usability, interoperability, and is capable of supporting multiple languages. Once the requirements were specified and the engineering methodology clarified, a unified service creation framework, called *i*SemServ (*i*ntelligent *sem*antic *serv*ices), was designed.

The framework was designed by following a multi-layered approach, where each identified building block occupies its own layer, but is mapped to other building blocks in a loosely coupled fashion. This enables the building blocks to exist independently of each other, and to be able to interoperate without noticeable restrictions. The key towards simplifying and accelerating the process of building intelligent semantic services was the choice of a model-driven approach, which forms part of the identified design requirements.

Further to this, requirements, such as complexity hiding, and re-usability, directly addressed the question of how to simplify and accelerate the engineering process. The intelligence aspects inscribed within the intelligent semantic service definition were conceptually addressed, based on the ontology-based and agent-based design requirements.

SQ4: *How can we implement the specified service creation framework in a unified and scalable environment?*

In this supporting question, the goal was to determine the manner in which the proposed service creation framework could be efficiently implemented, so as to adequately address the design requirements. The analysis of different implementation environments revealed that the framework could be effectively implemented within an open and extensible development environment, such as Eclipse, which exhibits a number of benefits, as discussed in Chapter 7 (Section 7.1 – 7.3).

Eclipse, as an SOA-based development environment, enables the easier integration of different technologies; and it provides a platform for creating singleton plug-ins that immediately interoperate with other Eclipse plug-ins. Thus, all the layers of the *i*SemServ framework were implemented within the Eclipse environment.

A number of open source tools were exploited in Eclipse to achieve the implementation of different layers. These included tools, such as: (1) UML2 SDK for designing service models; (2) Acceleo platform for defining model transformation rules, and code generation templates; (3) WSML editors for reviewing and editing generated domain ontologies; and (4) a JADE platform for developing all the defined intelligent building blocks. In the end, it became evident that Eclipse was a good choice for implementing our proposed solution, as all the layers were implemented by using a singleton Eclipse plug-in, as demonstrated in Chapters 7 and Chapter 8, and most of the design requirements were addressed without any major restrictions.

SQ5: *How can we evaluate the overall proposed solution for validity and relevance?*

In order to validate the plausibility and the relevance of the proposed solution, a number of techniques were adopted, as discussed in Chapter 1 (Section 1.6) and Chapter 8. In particular, qualitative and quantitative approaches were exploited. Using a use case scenario, the functionality and utility of all the layers in the *i*SemServ framework were demonstrated. This was followed by a comparative analysis, which provided a setting for qualitatively gauging our proposed solution with existing solutions in literature.

In the analysis, the key differentiators between our solution and related solutions were highlighted and clarified. From the analysis, it became evident that the value propositions of our framework are:

- **Uniformity:** providing and end-to-end approach of engineering intelligent semantic services, thus enabling the developer to use one platform to realize all the modules comprising such services.
- **Model-driven:** enabling average and expert service engineers to focus on developing intelligent semantic services in a structured, extensible, and platform-independent manner. Thus, increasing developers' productivity and minimizing development and maintenance costs.
- **Complexity hiding** in the form of automatic code generators supporting different architectural styles and semantic models by exploiting template-based code generators.
- **Intelligence wrapping** of services at message and ontological levels for the purposes of automatically processing semantic service requests and responses: in addition to reasoning over domain ontologies and semantic descriptions. JADE implements the collaborative and autonomous properties, and JESS implements the proactive and reactive properties (dealing with the reasoning capabilities using JESS rules. This ensures that the intelligent

semantic service developed according to the *i*SemServ framework conforms to the properties discussed in Chapter 5 (cf. Section 5.2).

The *i*SemServ framework was further evaluated, using the SEALS methodology – specifically meant for evaluating semantic technologies. We evaluated the solution on *performance*, that is, in terms of automatically generating different code skeletons, and *scalability*, that is, in terms of the support for an increasing size of service model, syntactic services, service descriptions, ontologies, semantic descriptions, and intelligence.

The evaluation activity demonstrated that the *i*SemServ framework is capable of handling an increasing service model size. Furthermore, the amount of time it takes to generate all the necessary intelligent semantic services modules is smaller when compared with the amount of time that the service engineer would take to manually generate all the code involved in building intelligent semantic services.

9.4. RESEARCH CONTRIBUTIONS

In Chapter 1 (Section 1.5), the primary and secondary contributions emanating from this study were highlighted. In this section, the focus is mainly on the key contributions that became apparent from the proposal and the practical implementation of the conceptual service framework.

The following are the noticeable research contributions forthcoming from this thesis:

- A clear definition of what is meant by an *intelligent semantic service.*
- Fundamental building blocks that comprise intelligent semantic services and their characteristics.
- A model-driven engineering methodology, based on software, Web, and service engineering philosophies for building intelligent semantic services.
- Essential requirements for designing and developing a unified service creation framework in a platform-independent manner.

- A unified, model-driven, and multi-layered *i*SemServ framework for addressing some of the challenges involved in developing intelligent semantic services.

Overall, the proposed *i*SemServ solution succeeds in simplifying and accelerating the process of engineering intelligent semantic services. It is a simple, and yet useful, approach for enabling average and expert service engineers to focus on developing semantic services in a structured, extensible, and unified manner.

9.5. RESEARCH LIMITATIONS

Overall, the objectives set out in Chapter 1 were accomplished. However, the proposed solution could still be improved to address some of the limitations identified during the implementation and evaluation phase. Some of these limitations are briefly discussed as follows:

- The ***multiple language support*** feature is still limited, in the sense that it depends on UML meta-models, which are language-dependent.
- ***Code generations,*** as proposed in our solution, do address the issue of complexities surrounding the building of semantic descriptions and ontologies. However, our solution is limited, in the sense that once the skeleton code for different layers is generated, the developer still has the obligation to understand the generated code. This is even more essential in cases where the code need to be augmented or edited. Furthermore, the generation of rules for reasoning purposes is limited in a sense that only templates are generated. The developer still has to manually define the specific rules for each intelligent semantic service.
- ***Agent-based intelligence,*** as proposed in this thesis, might be appropriate, but in some cases, it can prove to be a limitation. This is due to the fact that software agents have unaddressed challenges, such as security, incompatible messaging protocols, and resource-constraint limitations. Similar to what has been highlighted above, the generation of JESS rules is labour intensive as manual input from the developer is still required.

9.6. FURTHER RESEARCH

The potential for further research presented in this area is derived from the limitations of the proposed solution, as described in the previous section.

9.6.1. Improving Code-Generation Techniques

Currently, the *i*SemServ platform implements a model-to-code transformation module by exploiting different templates developed using Acceleo and UML service models made up of class diagrams. Further work in improving this technique could focus on enabling the code-generation module to also transform activity or sequence diagrams to different building blocks. This could further improve complexity hiding to the extent that, for example, activity diagrams could be automatically transformed to partial, and yet useful, syntactic services logic. In its current implementation, the *i*SemServ platform is only capable of generating code skeletons from class diagrams.

The service engineer is still required to manually complete the implementation of all the logic behind syntactic services, which could be modelled using activity and sequence diagrams. In addition, the intelligence layer also requires the service engineer to augment the generated intelligent skeletons, especially for each generated service provider agent that might have additional requirements not initially annotated within the UML service model. However, with regard to the generation of domain ontologies, semantics, descriptions, and deployment descriptors, the service developer is not required to implement additional logic for the generated artefacts.

9.6.2. Extending the Multiple Language Support Feature

In its current implementation, the *i*SemServ framework focuses on UML meta-models to accomplish the multiple language support feature. Nonetheless, this is limited, as discussed in Section 9.3. Moreover, finding a solution to such a limitation is not simple. Thus, further work could be done on proposing other innovative means to enable multi-language support when engineering intelligent semantic services in a unified platform, such as the *i*SemServ platform.

9.6.3. Enhancing the Intelligence Layer

The weaknesses of software agents, for instance, incompatible and proprietary messaging protocols, as described in the thesis, also call for further improvements to the intelligence layer. Our proposed solution attempts to address this limitation by mapping services, semantics and agents at different levels of abstraction. That is, at the knowledge-level using ontologies for semantic descriptions and agents; and at the message-level using open standard protocols, such as HTTP for services and agents integration.

Future work could focus on mapping services, semantic descriptions, and intelligence at one level, using approaches that address the current limitations of software agents effectively.
Finally, our solution could be extended and improved by incorporating all the phases involved in the engineering processes of intelligent semantic services, such as service discovery, service selection, service composition, and service monitoring.

APPENDICES

APPENDIX A: ABSTRACTS OF PUBLICATIONS

The following are the abstracts of the publications that emanated from this thesis, as listed in Chapter 1 (Section 1.8)

Towards a service creation framework: a case of intelligent semantic services

Abstract. Semantic Web Services are touted as one possible solution for some of the challenges experienced with Web services; such as lack of automatic service discovery and consumption. Ideally, semantic services are meant to facilitate automatic business service provisioning and consumption on the Web. These services are enriched with semantics, which are derived from ontologies. Nevertheless, semantic-based services are seldom adopted and utilised by service providers and consumers, respectively.

Some of the reasons noted in literature for this lack of adoption and usage include issues, such as the lack of real-life prototypes that are meant to demonstrate the benefits of semantic services; the lack of integrated service creation frameworks; unified development platforms that are purported to guide and promote simple engineering of semantic services. Thus, in this short paper, our aim is to propose and present of a conceptual multi-layered, and yet integrated, service creation framework – called *i*SemServ. The framework is intended to guide, simplify, and accelerate the process of engineering intelligent semantic services.

***i*SemServ: Towards the Engineering of Intelligent Semantic-Based Services**

Abstract. The emergence of Semantic Web Services is stimulating the need for modern enterprises to efficiently and rapidly develop and deliver machine-processable and machine-interpretable value-added services, in order to automate a variety of tasks on the Web. However, semantic-based services are seldom adopted and utilised, as there are few real-life examples that demonstrate the possibilities and benefits of such services. Furthermore, there is a lack of service creation frameworks and technical platforms that purport to guide and promote the simple, flexible, rapid, and unified engineering of semantic-based services.

In addition, current semantic service platforms do not support the construction of semantic services that are intelligent beyond the application of ontologies. In this paper, preliminary efforts that seek to address the challenges of simplifying and speeding up the engineering process of intelligent semantic services are presented. The goal of the work presented in this paper is to supply service providers, designers, and consumers with simple, unified, and yet simple, tools that can aid in the technical implementation of intelligent semantic-based services.

The main contribution envisaged from this research is a conceptual service-creation framework, called *i*SemServ, and a technological service-creation platform, which is intended to simplify and support the phases of building intelligent semantic services in an integrated manner. The proposed research adopts a quantitative approach with the main focus on model-building, prototypes, and laboratory experiments.

Towards the engineering of intelligent semantic-based services building blocks and methodology

Abstract. Semantic-based services are emerging as phenomena that enable innovative broad provisioning and consumption of business services on the Web. Therefore, service providers often require flexible technological tools and platforms that facilitate and support the effective development and advertisement of such services. Similarly, service consumers need access to tools and platforms that would enable the seamless discovery and consumption of these services. However, within the semantic Web service domain, there is a lack of development platforms and tools that promotes effective, rapid, simple, and flexible engineering and deployment of intelligent semantic-based services (*IsS*).

This is quite apparent by the limited research focusing on the practical development of semantic-based services. In this paper, we propose and motivate that open and flexible engineering of IsS is of significant importance, particularly to service providers and consumers in developing economies; where software costs, development costs, and technical skills still remain a challenge.

As a work in progress, our main focus in this paper is on the basic fundamental building blocks: the elementary components that make up a functional IsS. A proposed service engineering approach for constructing an IsS is also detailed. The future outlook of our work is on the proposal and the realization of a technical framework and integrated development environment for IsS engineering.

Engineering RESTful semantic services on the fly

Abstract. Real-world implementations of semantic services that could enable seamless integration of heterogeneous and legacy IT systems on the fly are deficient. This could be attributed to the complexity of heavy-weight semantic technologies, which mostly have a steep learning curve. As a consequence, the evolution of modern approaches that purport to simplify the engineering of such services is a necessity. In this short paper, we present a work-in-progress model-driven approach that seeks to simplify and speed up the process of engineering RESTful semantic services.

The suggested approach promotes the automatic transformation of platform-independent service models to partial service implementation and semantic descriptions, in order to realize functional RESTful semantic services.

iSemServ: Facilitating the Implementation of Intelligent Semantic Services

Abstract

The process of developing semantic services is viewed by service developers as being complex, and tedious. The main barriers that have been identified include a steep learning curve for emerging semantic models and ontological languages, the lack of integrated tool support for developing semantic services, and lack of interoperability between emerging semantic technologies and matured Web service technologies. In addition, current efforts that are meant to ease the implementation of semantic services are fragmented; that is, developers are required to use a combination of disconnected tools to realize semantic services. Moreover, existing semantic technologies are tightly coupled to specific semantic models and service architectural styles; leading to restrictive development environments. In this paper, an iSemServ framework is proposed, and implemented as an Eclipse plug-in with the core objective to facilitate, unify, and accelerate the process of developing intelligent semantic services using semantic models and service architectural styles of choice. Experimental evaluations demonstrate that a solution, such as iSemServ has the potential to minimize some of the barriers associated with building intelligent semantic services.

APPENDIX B: TRANSFORMATION TEMPLATES

In this section, some of the scripts used for code-generation purposes at different layers are included. These are included to assist the reader in understanding the possible implementation of the different layers that make up the proposed service creation framework. These fragments of code are not meant to demonstrate the complete functional logic of the *i*SemServ platform.

Services Layer

```
<%
metamodel http://www.eclipse.org/uml2/2.1.0/UML
import org.acceleo.modules.uml2.services.Common
import org.acceleo.modules.uml2.services.ListServices
import org.acceleo.modules.uml2.services.StringServices
import org.acceleo.modules.uml2.services.Uml2Services
%>
<%script type="Class" name="fullFilePath"%>
<%if (hasStereotype("RESTful")){%>
/src/<%package.name.toPath()%>/<%name%>.java
<%}%>
<%if (hasStereotype("SOAP")){%>
/src/<%package.name.toPath()%>/<%name%>.java
<%}%>
<%if (!hasStereotype("SOAP")&&!hasStereotype("RESTful")){%>
/src/<%package.name.toPath()%>/<%name%>.java
<%}%>
<%script type="uml.Class" name="rest" file="<%fullFilePath%>"%>
<%if (hasStereotype("RESTful")){%>
/*---------------------------------------------------------------------------------------------
 * <auto-generated>
 * Generated by iSemServ Model2Service transformer using Acceleo 2.7
 * Copyright (c) 2011 iSemServ
 *
 * All rights reserved.  This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License 1.0
 * You can apply any license to the files generated with this template
 * Original template generator contributor : Jabu Mtsweni, SAP Research, Pretoria, South Africa
 * <auto-generated>
 * ---------------------------------------------------------------------------------------------*/

package <%package.name%>;

/*
* JAX-RS imports
*/

import javax.ws.rs.*;

<%for (getAssociations().filter("Association").oppositeAttributeOf(current())[isNavigable()]){%>
<%if (current("Class").package!=type.){%>
import <%package.name%>.<%name%>;
<%}%><%}%>
<%if (superClass.nSize()==1){%>
import <%package.name%>.<%general.name%>;
<%}%>

/**
 * @author <Include your name>
 * @Date Created: <%getLongDate()%> [<%getTime()%>]
 */

 /**
 * @Path
 * represents relative URI for a RESTful resource
 */

 @Path("/<%name.toLowerCase()%>")
<%if (superClass.nSize()==1){%>
<%visibility%> class <%name%> extends <%general.name%><%}else{%><%visibility%> class
<%name%><%}%>
  {
  /*
 * @Declaration of Attributes
 */
<%if (attribute.nSize==0){%>
 //No attributes declared
<%}else{%>
<%for (attribute){%>
<%visibility%><%type.name%><%name%><%if (default!=null){%>="<%default%>"<%}%>;
<%}%>
<%}%>
 /*
 * @Declaration of Operations
 */
<%-- Generate methods -----------------------------------------------------%>
<%for (ownedOperation[!name.equalsIgnoreCase(current(1).name)].sep("\n")){%>
<%--** Generate methods doc ------------------------------------------------%>
/*
 * Description of the method <%name%>
 *
      <%for (ownedParameter[direction != "return"]){%>
 * @param <%name%>
      <%}%>
      <%for (ownedParameter[direction == "return"]){%>
 * @return <%type.name%>
      <%}%>
 */
 /**
 * decorate our RESTful service with @Path, @HTTP_Method, and @Representation
 */
<%if (name.startsWith("get")){%>
  @Path("/<%name.toLowerCase()%>")
  @GET
  @Consumes({"text/plain","application/xml","text/html","application/json"})
  @Produces({"text/plain","application/xml","text/html","application/json"})
<%if (type.name!=null) {%>
<%visibility%><%type.name%><%name%>(@PathParam("<%ownedParameter.name.sep("
")%>")<%ownedParameter[!direction.equalsIgnoreCase("return")].parameterDeclaration.sep(",")%>){

  //TODO: ADD service logic for <%name%> method

<%if (type.name!=null){%>
  return null;
<%}else{%>
<%--Does not return anything--%>
<%}%>
  }
<%}else{%>
<%visibility%> void
<%name%>(<%ownedParameter[!direction.equalsIgnoreCase("return")].parameterDeclaration.sep(",")%>)
 {

  //<%startUserCode%>

   //TODO: ADD service logic for <%name%> method

  //<%endUserCode%>
```

```

<%if (type.name!=null){%>
  return null;
<%}else{%>
<%--Does not return anything--%>
<%}%>
  }
<%}%>
<%}%>
<%if (name.startsWith("request")){%>
 @Path("/<%name.toLowerCase()%>")
 @GET
 @Consumes({"text/plain","application/xml","text/html","application/json"})
 @Produces({"text/plain","application/xml","text/html","application/json"})
<%if (type.name!=null) {%>
<%visibility%><%type.name%><%name%>(@PathParam("<%ownedParameter.name.sep(" ")%>")<%ownedParameter[!direction.equalsIgnoreCase("return")].parameterDeclaration.sep(",")%>){

 //TODO: ADD service logic for <%name%> method

<%if (type.name!=null){%>
  return null;
<%}else{%>
<%--Does not return anything--%>
<%}%>
  }
<%}else{%>
<%visibility%> void
<%name%>(<%ownedParameter[!direction.equalsIgnoreCase("return")].parameterDeclaration.sep(",")%>)
 {

  //<%startUserCode%>

   //TODO: ADD service logic for <%name%> method

  //<%endUserCode%>

<%if (type.name!=null){%>
  return null;
<%}else{%>
<%--Does not return anything--%>
<%}%>
  }
<%}%>
<%}%>
<%if (name.startsWith("update")){%>
 @Path("/<%name.toLowerCase()%>")
 @PUT
 @Consumes({"text/plain","application/xml","text/html","application/json"})
 @Produces({"text/plain","application/xml","text/html","application/json"})
<%if (type.name!=null) {%>
<%visibility%><%type.name%><%name%>(@PathParam("<%ownedParameter.name.sep(" ")%>")<%ownedParameter[!direction.equalsIgnoreCase("return")].parameterDeclaration.sep(",")%>){

 //TODO: ADD service logic for <%name%> method

<%if (type.name!=null){%>
  return null;
<%}else{%>
<%--Does not return anything--%>
<%}%>
  }
<%}else{%>
<%visibility%> void
<%name%>(<%ownedParameter[!direction.equalsIgnoreCase("return")].parameterDeclaration.sep(",")%>)
 {

  //<%startUserCode%>

   //TODO: ADD service logic for <%name%> method

  //<%endUserCode%>

<%if (type.name!=null){%>
  return null;
<%}else{%>
<%--Does not return anything--%>
<%}%>
  }
<%}%>
<%}%>
<%if (name.startsWith("create")){%>
 @Path("/<%name.toLowerCase()%>")
 @POST
 @Consumes({"text/plain","application/xml","text/html","application/json"})
 @Produces({"text/plain","application/xml","text/html","application/json"})
<%if (type.name!=null) {%>
<%visibility%><%type.name%><%name%>(<%ownedParameter[!direction.equalsIgnoreCase("return")].parameterDeclaration.sep(",")%>){

 //TODO: ADD service logic for <%name%> method

<%if (type.name!=null){%>
  return null;
<%}else{%>
<%--Does not return anything--%>
<%}%>
  }
<%}else{%>
<%visibility%> void <%name%>(<%ownedParameter[!direction.equalsIgnoreCase("return")].parameterDeclaration.sep(",")%>)
 {

  //<%startUserCode%>

   //TODO: ADD service logic for <%name%> method

  //<%endUserCode%>

<%if (type.name!=null){%>
  return null;
<%}else{%>
<%--Does not return anything--%>
<%}%>
  }
<%}%>
<%}%>
<%if (name.startsWith("add")){%>
 @Path("/<%name.toLowerCase()%>")
 @POST
 @Consumes({"text/plain","application/xml","text/html","application/json"})
 @Produces({"text/plain","application/xml","text/html","application/json"})
<%if (type.name!=null) {%>
<%visibility%><%type.name%><%name%>(<%ownedParameter[!direction.equalsIgnoreCase("return")].parameterDeclaration.sep(",")%>){

 //TODO: ADD service logic for <%name%> method

<%if (type.name!=null){%>
  return null;
<%}else{%>
<%--Does not return anything--%>
<%}%>
  }
```

```
<%}else{%>
<%visibility%> void <%name%>(<%ownedParameter[!direction.equalsIgnoreCase("return")].parameterDeclaration.sep(",")%>)
 {

 //<%startUserCode%>

  //TODO: ADD service logic for <%name%> method

 //<%endUserCode%>

<%if (type.name!=null){%>
 return null;
<%}else{%>
<%--Does not return anything--%>
<%}%>
 }
<%}%>
<%}%>
<%if (name.startsWith("delete")){%>
@Path("/<%name.toLowerCase()%>/{<%ownedParameter[!direction.equalsIgnoreCase("return")].parameterDeclaration.toLowerCase().sep(",")%>}")
 @DELETE
 @Consumes({"text/plain","application/xml","text/html","application/json"})
 @Produces({"text/plain","application/xml","text/html","application/json"})
<%if (type.name!=null) {%>
<%visibility%><%type.name%><%name%>(@PathParam("<%ownedParameter.name.toLowerCase().sep(" ")%>")<%ownedParameter[!direction.equalsIgnoreCase("return")].parameterDeclaration.sep(",")%>){

 //TODO: ADD service logic for <%name%> method

<%if (type.name!=null){%>
 return null;
<%}else{%>
<%--Does not return anything--%>
<%}%>
 }
<%}else{%>
<%visibility%> void <%name%>(<%ownedParameter[!direction.equalsIgnoreCase("return")].parameterDeclaration.sep(",")%>)
 {

 //<%startUserCode%>

  //TODO: ADD service logic for <%name%> method

 //<%endUserCode%>

<%if (type.name!=null){%>
 return null;
<%}else{%>
<%--Does not return anything--%>
<%}%>
 }
<%}%>
<%}%>
<%}%>
}
<%}%>
<%-- end of rest --%>
<%if (hasStereotype("SOAP")){%>
/*-----------------------------------------------------------------------------------------------
 * <auto-generated>
 * Generated by iSemServ Model2Service transformer using Acceleo 2.8
 * Copyright (c) 2011 iSemServ
 *
 * All rights reserved.  This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License 1.0
 * You can apply any license to the files generated with this template
 * Original template generator contributor : Jabu Mtsweni, SAP Research, Pretoria, South Africa
 * <auto-generated>
 * ---------------------------------------------------------------------------------------------/
package <%package.name%>

/**
 * @author <Include your name>
 * @Date Created: <%getLongDate()%> [<%getTime()%>]
 */
/*
* JAX-WS imports
*/
import javax.jws.WebMethod;
import javax.jws.WebService;

/**
* @WebService
* Important for decorating our class as a SOAP Web Service
*/
 @WebService(serviceName = "<%name%>",
                              portName = "<%name%>Port",
                              endpointInterface = "<%package.name%>.<%name%>Interface",
                              targetNamespace = "http://<%package.name.reverse()%>",
                              wsdlLocation=" WebContent/wsdl/<%name.toLowerCase()%>.wsdl")
<%visibility%> class <%name%> {

  /**
 * @Declaration of Attributes
 */
<%if (attribute.nSize==0){%>
 //No attributes declared
<%}else{%>
<%for (attribute){%>
<%visibility%><%type.name%><%name%>;
<%}%>
<%}%>

 /**
 * @Declaration of Operations
 */
<%-- Generate methods -----------------------------------------------------%>
<%for (ownedOperation[!name.equalsIgnoreCase(current(1).name)].sep("\n")){%>
<%--** Generate methods doc ------------------------------------------------%>
/**
 * Description of the method <%name%><%ownedComment%>.
 *
      <%for (ownedParameter[direction != "return"]){%>
 * @param <%name%><%ownedComment%>
      <%}%>
      <%for (ownedParameter[direction == "return"]){%>
 * @return <%name%><%ownedComment%>
      <%}%>
 */
<%if (type.name!=null) {%>
  @WebMethod
<%visibility%><%type.name%><%name%>(<%ownedParameter[!direction.equalsIgnoreCase("return")].parameterDeclaration.sep(",")%>)
  {

  //TODO-logic for <%name%> method

<%if (type.name!=null){%>
   return null;
<%}else{%>
<%--Does not return anything--%>
<%}%>
  }
<%}else{%>
  @WebMethod
<%visibility%> void <%name%>(<%ownedParameter[!direction.equalsIgnoreCase("return")].parameterDeclaration.sep(",")%>)
 {
 //<%startUserCode%>

 //ADD service logic for <%name%> method

 //<%endUserCode%>

<%if (type.name!=null){%>
  return null;
<%}else{%>
<%--Does not return anything--%>
<%}%>
  }
<%}%>
<%}%>
}
<%}%>
<%--end of the soap if--%>
<%-- Beginning of POJO classes --%>
<%if (!hasStereotype("SOAP")&&!hasStereotype("RESTful")) {%>
/*------------------------------------------------------------------------------------------
 * <auto-generated>
 * Generated by SemServ Model2Service transformer using Acceleo 2.7
 * Copyright (c) 2011 iSemServ
 *
 * All rights reserved.  This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License 1.0
 * You can apply any license to the files generated with this template
 * Original template generator contributor : Jabu Mtsweni, SAP Research, Pretoria, South Africa
 * <auto-generated>
 * ------------------------------------------------------------------------------------------*/

package <%package.name%>;

/**
 * @author <Include your name>
 * @Date Created: <%getLongDate()%> [<%getTime()%>]
 * @category <%name%>Entity
 */
<%--Start of POJO class --%>
<%if (superClass.nSize()==1){%>
<%visibility%> class <%name%> extends <%general.name%><%}else{%><%visibility%> class <%name%><%}%>
 {
 /*
 * @Declaration of Attributes
 */
<%if (attribute.nSize==0){%>
//No attributes declared
<%}else{%>
<%for (attribute){%>
<%visibility%><%type.name%><%name%><%if (default!=null){%>="<%default%>"<%}%>;
<%}%>
<%}%>
/*
 * @Declaration of Operations
 */
<%-- Generate methods -----------------------------------------------------%>
<%for (ownedOperation[!name.equalsIgnoreCase(current(1).name)].sep("\n")){%>
<%--** Generate methods doc -----------------------------------------------%>
/**
```

```
* Description of the method <%name%><%ownedComment%>.
*
    <%for (ownedParameter[direction != "return"]){%>
* @param <%name%><%ownedComment%>
    <%}%>
    <%for (ownedParameter[direction == "return"]){%>
* @return <%name%><%ownedComment%>
    <%}%>
*/
<%if (type.name!=null) {%>
<%visibility%><%type.name%><%name%>(<%ownedParameter[!direction.equalsIgnoreCase("return")].parameterDeclaration.sep(",")%>)
 {

 //TODO: ADD service logic for <%name%> method

<%if (type.name!=null){%>
  return null;
<%}else{%>
<%--Does not return anything--%>
<%}%>
  }
<%}else{%>
<%visibility%> void <%name%>(<%ownedParameter[!direction.equalsIgnoreCase("return")].parameterDeclaration.sep(",")%>)
 {
 //<%startUserCode%>

 //TODO: ADD service logic for <%name%> method

 //<%endUserCode%>

<%if (type.name!=null){%>
  return null;
<%}else{%>
<%--Does not return anything--%>
<%}%>
  }
<%}%>
<%}%>
}
<%}%>
<%scripttype="Parameter" name="parameterDeclaration"%>
<%type.name%><%name%>
```

DEPLOYMENT

```
<%
metamodel http://www.eclipse.org/uml2/2.1.0/UML
import org.acceleo.modules.uml2.services.Common
import org.acceleo.modules.uml2.services.ListServices
import org.acceleo.modules.uml2.services.StringServices
import org.acceleo.modules.uml2.services.Uml2Services
%>
<%scripttype="uml.Class" name="fullFilePath"%>
<%if hasStereotype("RESTful") {%>
/WebContent/WEB-INF/web.xml
<%}%>
<%if hasStereotype("SOAP"){%>
/WebContent/WEB-INF/web.xml
<%}%>
<%scripttype="uml.Class" name="rest" file="<%fullFilePath%>"%>
<%if hasStereotype("RESTful") {%>
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
<display-name><%package.name%></display-name>
<welcome-file-list>
<welcome-file>index.html</welcome-file>
<welcome-file>index.htm</welcome-file>
<welcome-file>index.jsp</welcome-file>
<welcome-file>default.html</welcome-file>
<welcome-file>default.htm</welcome-file>
<welcome-file>default.jsp</welcome-file>
</welcome-file-list>
<servlet>
<servlet-name>Jersey REST <%name.toLowerCase()%>Service</servlet-name>
<servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>Jersey REST Service</servlet-name>
<url-pattern>/*</url-pattern>
</servlet-mapping>
</web-app>
<%}%>
<%if hasStereotype("SOAP") {%>
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems,
Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<web-app>
<listener>
<listener-class>
com.sun.xml.ws.transport.http.servlet.WSServletContextListener
</listener-class>
</listener>
<servlet>
<servlet-name><%name.toLowerCase()%></servlet-name>
<servlet-class>
com.sun.xml.ws.transport.http.servlet.WSServlet
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name><%name.toLowerCase()%></servlet-name>
<url-pattern>/<%name.toLowerCase()%></url-pattern>
</servlet-mapping>
<session-config>
<session-timeout>120</session-timeout>
</session-config>
</web-app>
<%}%>
```

Semantics Layer

```
<%
metamodel http://www.eclipse.org/uml2/2.0.0/UML
import org.acceleo.modules.uml2.services.Uml2Services
import org.acceleo.modules.uml2.services.Common
%>
<%--This template generate a WSMO domain ontologies--%>
<%script type="Class" name="fullFilePath"%>
<%if (hasStereotype("WSMO")){%>
wsml/ontologies/<%name.toLowerCase()%>Ontology.wsml
<%}%>
<%script type="Class" name="ontologies" file="<%fullFilePath%>"%>
<%if (hasStereotype("WSMO")){%>
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"
//<!--Generated by SemServ Model2Semantics transformer using Acceleo 2.8-->
//<!--Date: <%getLongDate()%> [<%getTime()%>] -->
namespace {_"http://www.isemserv.co.za/ontologies/<%name.toLowerCase()%>Ontology.wsml#",
                    dc _"http://purl.org/dc/elements/1.1#",
                    xsd _"http://www.w3c.org/2001/XMLSchema#",
                    wsml _"http://www.wsmo.org/2004/wsml-syntax#",
              dt _"http://www.wsmo.org/ontologies/dateTime/#",
      desc _"http://www.isemserv.co.za/descriptions#"}


ontology _"http://www.isemserv.co.za/wsml/ontologies/<%name.toLowerCase()%>Ontology.wsml#"

nonFunctionalProperties
  dc#type hasValue "<%name.toLowerCase()%> Domain Ontology"
  dc#description hasValue "Enter description of the domain ontology"
  dc#title hasValue "Domain Ontology for a <%name.toLowerCase()%> Web service"
  dc#creator hasValue {"Your Name/Editors name"}
  dc#subject hasValue { "<%name.toU1Case()%>", "{other subjects}"}
  dc#publisher hasValue "iSemServ"
  dc#date hasValue "<%getLongDate()%> [<%getTime()%>]"
  dc#type hasValue _"http://www.wsmo.org/2004/d2#ontologies"
  dc#identifier hasValue _"http://www.isemserv.co.za/ontologies/<%name.toLowerCase()%>Ontology"
  dc#language hasValue "en-US"
  dc#format hasValue "text/plain"
  endNonFunctionalProperties

importsOntology {_"http://example.org/ImportedOntology"}

concept <%name%>
nonFunctionalProperties
dc#description hasValue "{add description of the concept}"
endNonFunctionalProperties
<%for (ownedAttribute) {%>
<%name%> ofType _<%type.name%>
<%}%>
<%for (ownedOperation) {%>

concept <%name%>
nonFunctionalProperties
dc#description hasValue "{add description of the concept}"
endNonFunctionalProperties
<%for (ownedParameter[!direction.equalsIgnoreCase("return")]) {%>
              <%name%> ofType _<%type.name%>
<%}%>
<%}%>
```

WEB SERVICES (WSMO)

```
<%
metamodel http://www.eclipse.org/uml2/2.0.0/UML
import org.acceleo.modules.uml2.services.Uml2Services
import org.acceleo.modules.uml2.services.Common
%>
<%--
This template generate a Web Service capability for WSMO
--%>
<%script type="Class" name="fullFilePath"%>
<%if (hasStereotype("WSMO")){%>
wsml/services/<%name.toLowerCase()%>WSCapability.wsml
<%}%>
<%script type="Class" name="ontologies" file="<%fullFilePath%>"%>
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"
comment <!--Generated by SemServ Model2Semantics transformer using Acceleo 2.8-->
comment <!--Date: <%getLongDate()%> [<%getTime()%>] -->
namespace { _"http://www.isemserv.co.za/services/<%name.toLowerCase()%>Semantics#",
<%name.toLowerCase().substring(0,3)%> _"http://www.isemserv.co.za/ontologies#",
dc _"http://purl.org/dc/elements/1.1#",
wsml _"http://www.wsmo.org/wsml/wsml-syntax#",
xsd  _"http://www.w3.org/2001/XMLSchema#",
desc _"http://www.isemserv.co.za/descriptions#"}
webService <%ownedOperation.name%>Service
importsOntology {_"http://example.org/ImportedOntology"}  /*PLEASE COMPLETE*/
capability <%ownedOperation.name%>Capability
nonFunctionalProperties
dc#type hasValue "service ontology"
```

```
dc#description hasValue "Enter description for this capability"
dc#title hasValue "Capability for a <%name.toLowerCase()%> Web service"
dc#creator hasValue {"Your Name"}
dc#publisher hasValue "isemserv"
dc#date hasValue "<%getLongDate()%> [<%getTime()%>]"
dc#type hasValue _"http://www.wsmo.org/2004/d2#ontologies"
dc#identifier hasValue _"http://www.isemserv.co.za/services/<%name.toLowerCase()%>"
dc#language hasValue "en-US"
dc#format hasValue "text/plain"
desc#serviceDescription hasValue "COMPLETE URL FOR SERVICE DESCRIPTION"
endNonFunctionalProperties

<%for (ownedOperation) {%>
sharedVariables {<%for (ownedParameter[!direction.equalsIgnoreCase("return")].sep(", "))
{%>?<%name%><%}%>}
<%}%>
precondition
nonFunctionalProperties
dc#description hasValue "condition(s) that need to be satisfied before service is invoked"
endNonFunctionalProperties
definedBy
<%for (ownedOperation) {%>
<%for (ownedParameter[!direction.equalsIgnoreCase("return")]) {%>
?<%name%> memberOf <%name.toU1Case()%>
<%}%>
                <%}%>
```

Intelligence Layer

```
<%
metamodel http://www.eclipse.org/uml2/2.1.0/UML
import org.acceleo.modules.uml2.services.Common
import org.acceleo.modules.uml2.services.ListServices
import org.acceleo.modules.uml2.services.StringServices
import org.acceleo.modules.uml2.services.Uml2Services
%>
<%script type="Class" name="fullFilePath"%>
<%if (hasStereotype("RESTful")){%>
/src/<%package.name.toPath()%>/agents/provider/<%name%>ProviderAgent.java
<%}%>
<%script type="uml.Class" name="ServiceAgent" file="<%fullFilePath%>"%>
<%if (hasStereotype("RESTful")){%>
/*----------------------------------------------------------------------------------------------
<auto-generated>
Generated by iSemServ Model2Intelligence transformer using Acceleo 2.8
Copyright (c) 2011 iSemServ
All rights reserved.  The generator and the accompanying materials
are made available under the terms of the Eclipse Public License 1.0
You can apply any license to the files generated with this template

ServiceProviderAgent template generator contributor : Jabu Mtsweni
UNISA 2011
JADE - Java Agent DEvelopment Framework is a framework to develop
multi-agent systems in compliance with the FIPA specifications.
Copyright (C) 2000 CSELT S.p.A.  GNU Lesser General Public License
<auto-generated>
----------------------------------------------------------------------------------------------*/

package <%package.name%>.agents.provider;
/**Required Libraries for the Intelligence Layer*/
import java.io.IOException;
import jade.core.Agent;
import jade.core.behaviours.CyclicBehaviour;
import jade.domain.DFService;
import jade.domain.FIPAException;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;
import isemserv.org.wadl.WadlReader;
import isemserv.org.wsml.WSMLReader;
/**
@co-author Jabu Mtsweni
@category ServiceProviderAgent
@version $Revision: 1.0
@Date:  <%getLongDate()%> [<%getTime()%>]
*/
public class <%name%>ProviderAgent extends Agent {

/**
default Agent properties
*/
private static final long serialVersionUID = 1L;
/**
Initialise ServiceProviderAgent
*/
protected void setup(){
//Logging "welcome message"
System.out.println("Hallo! :"+getAID().getName()+" is initialized and ready--->");
/**
Register ServiceProviderAgent in YellowPages (JADE)
*/
DFAgentDescription dfd = new DFAgentDescription();
dfd.setName(getAID());
ServiceDescription sd = new ServiceDescription();
sd.setType("<%name.toLowerCase()%>");
sd.setName("JADE-service-provider");
dfd.addServices(sd);
try {
DFService.register(this, dfd);
//Logging a confirmation  message for Registration
System.out.println(getAID().getName()+" is registered in JADE Yellow Pages");
}
catch (FIPAException fe) {
fe.printStackTrace();
}
/**
Add getRESTServiceURL Cyclic Behaviour provided by ServiceProviderAgent
*/
addBehaviour(new getServiceURL());
/**
Add getWSCapabilityName Cyclic Behaviour provided by ServiceProviderAgent
*/
addBehaviour(new getWSCapability());

/**
Add Generic Cyclic Behaviour provided by ServiceProviderAgent
*/
<%-- Generate methods ----------------------------------------------------%>
<%for (ownedOperation[!name.equalsIgnoreCase(current(1).name)].sep("\n")){%>
<%--** Generate methods doc ------------------------------------------------%>
/**
Add <%name%> Cyclic Behaviour provided by ServiceProviderAgent
*/
addBehaviour(new <%name%>());
<%}%>
```

JESS Rule Template

```
(defrule rule-name
"optional comment"
(pattern-1) ; left-hand side (LHS) of the rule
(pattern-2) ; consisting of elements before the "=>"
(pattern-n)
=>
(action-1) ; right-hand side (RHS) of the rule
(action-2) ; consisting of elements after the "=>"
(action-m)
) ; the last ")" balances the opening "(" to
; the left of "defrule". Be sure all your
; parentheses balance or you will get
; error messages.
```

JESS Template Syntax

```
(deftemplate classname
  (declare (from-class classname)
  (include-variables TRUE))
```

REFERENCES

ACCELEO. 2011. Acceleo - transforming models into code. [Online]. Available from: http://www.eclipse.org/acceleo/. [Accessed: 16 April 2011].

ACUNA, C. J. & MARCOS, E. 2006. Modeling semantic Web services: a case study. In: Proceedings of the 6th international conference on Web engineering (ICWE'06). 11-14 July. Palo Alto, California, USA.

AGARWAL, A., DASGUPTA, K., KARNIK, N., KUMAR, A., KUNDU, A., MITTAL, S., et al. 2005. A service creation environment based on end to end composition of web services. In: Proceedings of the WWW2005. 10-14 May. Chiba, Japan.

AGRE, G., MARINOVA, Z., PARIENTE, T. & MICSIK, A. 2007. Towards Semantic Web service engineering. In: Proceedings of the Workshop on service matchmaking and resource retrieval in the semantic Web (SMRR 2007).

AKKIRAJU, R., FARRELL, J., MILLER, J., NAGARAJAN, M., SCHMIDT, M.-T. & VERMA, A. S. K. 2005. Web Service Semantics - WSDL-S. [Online]. Available from: http://www.w3.org/Submission/WSDL-S/. [Accessed: 07 June 2009].

AL-FEEL, H., KOUTB, M. A. & SUOROR, H. 2009. Toward an agreement on Semantic Web architecture. *World academy of science, engineering, and technology,* 49, 806-810.

ALONSO, A., CASATI, F., KUNO, H. & MACHIRAJU, V. 2004. *Web Services: concepts, architectures, applications*: Springer.

ANABY-TAVOR., A., AMID., D., SELA., A., FISHER., A., ZHANG., K. & JUN., O. T. 2008. Towards a model driven service engineering process. In: Proceedings of the IEEE Congress on Services - Part I. 6-11 July.

AZIZ, Z., ANUMBA, C., RUIKAR, D., CARRILLO, P. & BOUCHLAGHEM, D. 2004. Semantic web based services for intelligent mobile construction collaboration. *ITcon,* 9, 367-369.

BACHLECHNER, D. 2008. Semantic Web service research: current challenges and proximate achievements. *International Journal of Computer Science and Applications,* 5(3b), 117-140.

BAIDA, Z., GORDIJN, J. & OMELAYENKO, B. 2004. A shared service terminology for online service provisioning. In: Proceedings of the Sixth International Conference on Electronic Commerce (ICEC'04). Delft, Netherlands.

BALACHANDRAN, B. M. 2008. Developing Intelligent Agent Applications with JADE and JESS. In: Proceedings of the 12th international conference on Knowledge-Based Intelligent Information and Engineering Systems, Part III. Zagreb, Croatia.

BALZER, S., LIEBIG, T. & WAGNER, M. 2004. Pitfalls of OWL-S: a practical semantic web use case. In: Proceedings of the 2nd international conference on Service oriented computing. 15-19 November. New York, NY, USA.

BATTLE, S., BERNSTEIN, A., BOLEY, H., GROSOF, B., GRUNINGER, M., HULL, R., et al. 2005. Semantic Web Services Framework (SWSF) overview. [Online]. Available from: http://www.w3.org/Submission/SWSF/. [Accessed: 25 March 2009].

BENSABER, D. A. & MALKI, M. 2008. Development of semantic web services: model driven approach. In: Proceedings of the 8th international conference on new technologies in distributed systems. Lyon, France.

BERNERS-LEE, T. 2000. Semantic Web - XML2000. [Online]. Available from: http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html. [Accessed: 04 June 2009].

BERNERS-LEE, T. 2003. The Semantic Web and challenges. [Online]. Available from: http://www.w3.org/2003/Talks/01-sweb-tbl/. [Accessed: 17 September 2009].

BERNERS-LEE, T. 2005. Web for real people. [Online]. Available from: http://www.w3.org/2005/Talks/0511-keynote-tbl/. [Accessed: 18 September 2009].

BERNERS-LEE, T. 2006. Artificial Intelligence and the Semantic Web. [Online]. Available from: http://www.w3.org/2006/Talks/0718-aaai-tbl/Overview.html. [Accessed: 23 October 2009].

BERNERS-LEE, T., FIELDING, R., IRVINE, U. C. & MASINTER, L. 2005. Uniform Resource Identifier (URI): generic syntax. [Online]. Available from: http://www.rfc-archive.org/getrfc.php?rfc=3986. [Accessed: 20 October 2009].

BERNERS-LEE, T., HENDLER, J. & LASSILA, O. 2001. The semantic web. *Scientific American*, 34-43.

BICER, V., LAMPARTER, S., SURE, Y. & DOGRU, A. H. 2009. Towards an interdisciplinary methodology for service-oriented system engineering. In: Proceedings of the 24th International Symposium on Computer and Information Sciences.

BIERMANN, E. 2004. *A framework for the protection of mobile agents against malicious hosts.* University of South Africa, Pretoria.

BLOIS, M., ESCOBAR, M. & CHOREN, R. 2007. Using agents and ontologies for application development on the semantic web. *Journal of Brazilian Computer Society,* 13(2), 35-44.

BOOTH, D., HAAS, H., MCCABE, F., NEWCOMER, E., CHAMPION, M. & FERRIS, C. 2004. Web services architecture. *W3C working group note* [Online]. Available from: http://www.w3.org/TR/ws-arch. [Accessed: 18 March 2009].

BOUCHIHA, D. & MALKI, M. 2010. Towards re-engineering Web applications into Semantic Web Services. In: Proceedings of the International Conference on Machine and Web Intelligence (ICMWI). 3-5 October.

BOUHISSI, H. E., MALKI, M. & BOUCHIHA, D. 2006. Towards WSMO ontology specification from existing Web Services. [Online]. Available from: http://ceur-ws.org/Vol-547/100.pdf. [Accessed: 18 October 2009].

BRAMBILLA, M., CELINO, I., CERI, S., CERIZZA, D., VALLE, E. D. & FACCA, F. M. 2006. Software engineering approach to design and development of semantic Web service applications. In: Proceedings of the 5th International Semantic Web Conference. 5-9 November. Athens, GA, USA

BREIVOLD, H. P. & LARSSON, M. 2007. Component-Based and Service-Oriented Software Engineering: key concepts and principles. In: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications. 28-31 August. Lubeck

BURSTEIN, M., BUSSLER, C., FININ, T., HUHNS, M. N., PAOLUCCI, M., SHETH, A. P., et al. 2005. A semantic Web services architecture. *IEEE Internet Computing,* 9(5), 72-81.

BUSSLER, C., ROMAN, D., LAUSEN, H., OREN, E. & LARA, R. 2004. Web Service Modeling Ontology - Lite (WSMO-Lite) In: Proceedings of the 1st F2F meeting SDK cluster working group on Semantic Web Services. 15 March. Wiesbaden, Germany.

CABRAL, L., DOMINGUE, J., MOTTA, E., PAYNE, T. & HAKIMPOUR, F. 2004. Approaches to semantic web services: an overview and comparisons. In: Proceedings of the European Semantic Web Conference. Heraklion, Greece.

CABRAL, L., DOMINGUE, J., GALIZIA, S., GUGLIOTTA, A., NORTON, B., TANASESCU, V., AND PEDRINACI, C. 2006. IRS-III: a broker for Semantic Web Services based applications. In: Proceedings of the 5th International Semantic Web Conference (ISWC 2006). 5-9 November. Athens, France.

CARDOSO, J. 2007a. *Semantic Web Services: theory, tools and applications*: IGI Global.

CARDOSO, J. 2007b. The Semantic Web vision: where are we? *IEEE Intelligent Systems,* 22(5), 84 - 88

CARDOSO, J., BARROS, A., MAY, N. & KYLAU, U. 2010. Towards a Unified Service Description Language for the Internet of Services: Requirements and First Developments. In: Proceedings of the IEEE International Conference on Services Computing (SCC). 5-10 July. Miami, Florida.

CARDOSO, J., VOIGT, K. & WINKLER, M. 2008. Service engineering for the internet of services. In: Proceedings of the Enterprise Information Systems 10th International Conference (ICEIS). 12-16 June. Barcelona, Spain.

CHEN, H.-M. 2008. Towards Service Engineering: Service Orientation and Business-IT Alignment. In: Proceedings of the 41st Annual Hawaii International Conference on System Sciences

CHINNICI, R., MOREAU, J.-J., RYMAN, A. & WEERAWARANA, S. 2007. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. [Online]. Available from: http://www.w3.org/TR/wsdl20/. [Accessed: 15 October 2009].

CHRISTENSEN, E., CURBERA, F., MEREDITH, G. & WEERAWARANA, S. 2001. Web Services Description Language (WSDL) 1.1. [Online]. Available from: http://www.w3.org/TR/wsdl. [Accessed: 12 March 2009].

CORCHO, O., GOMEZ-PEREZ, A., FERNANDEZ-LOPEZ, M. & LAMA, M. 2003. ODE-SWS: A semantic web service development environment. In: Proceedings of the 1st International Workshop on Semantic Web and Databases (SWDB03). Berlin, Germany.

CORCHO, O., SILVESTRE, L., BENJAMINS, R., BAS, J. L. & BELLIDO., S. 2007. Personal ebanking solutions based on semantic web services. *Studies in Computational Intelligence (SCI),* 37, 287 - 305.

COWLES, P. 2005. Web Service API and the Semantic Web. [Online]. Available from: http://soa.sys-con.com/node/39631. [Accessed: 17 August 2009].

DA SILVA, P., MCGUINNESS, D. & FIKES, R. 2006. A proof markup language for semantic web services. *Information Systems,* 31(4), 381-395.

DANNECKER, L., FELDMANN, M., NESTLER, T., HUBSCH, G., JUGEL, U., MUTHMANN, K., et al. 2010. Rapid Development of Composite Applications Using Annotated Web Services
Current Trends in Web Engineering. In (Vol. 6385: 1-12): Springer Berlin / Heidelberg.

DAVIS, F. D., BAGOZZI, R. P. & WARSHAW, P. R. 1989. User acceptance of computer technology: comparison of two theoretical models. *Management Science,* 35(8), 982-1003.

DE BRUIJN, J., BUSSLER, C., DOMINGUE, J., FENSEL, D., HEPP, M., KELLER, U., et al. 2005a. *Web Service Modelling Ontology (WSMO)*: DERI.

DE BRUIJN, J., FENSEL, D., KELLER, U. & LARA, R. 2005b. Using the web service modeling ontology to enable semantic e-business. *Communications of the ACM,* 48(12), 43-47.

DE BRUIJN, J., FENSEL, D., KERRIGAN, M., KELLER, U., LAUSEN, H. & SCICLUNA, J. 2008. *Modeling Semantic Web Services: The Web service modeling language*: Springer-Verlag Berlin Heidelberg.

DE BRUIJN, J., LAUSEN, H., KRUMMENACHER, R., POLLERES, A., PREDOIU, L., KIFER, M., et al. 2005c. *The Web Service Modeling Language WSML*: DERI.

DIMITROV, M., SIMOV, A., MOMTCHEV, V. & KONSTANTINOV, M. 2007. WSMO Studio - a semantic web services modelling environment for WSMO. In: Proceedings of the 4th European conference on the Semantic Web: research and applications. Innsbruck, Austria.

DOMINGUE, J., CABRAL, L., GALIZIA, S., TANASESCU, V., GUGLIOTTA, A., NORTON, B., et al. 2008. IRS-III: A broker-based approach to semantic Web services. *Web Semantics: Science, Services and Agents on the World Wide Web,* 6(2), 109-132.

DOMINGUE, J., CABRAL, L., HAKIMPOUR, F., SELL, D. & MOTTA, E. 2004. Demo of IRS-III: A platform and infrastructure for creating wsmo-based semantic web services. In: Proceedings of the 3rd International Semantic Web Conference (ISWC2004). 7-11 November. Hiroshima, Japan.

DUMEZ, C., NAIT-SIDI-MOH, A., GABER, J. & WACK, M. 2008. Modeling and Specification of Web Services Composition Using UML-S. In: Proceedings of the 4th International Conference on Next Generation Web Services Practices. 20-22 October. Seoul.

EITER, T., IANNI, G., KRENNWALLNER, T. & POLLERES, A. 2008. Rules and Ontologies for the Semantic Web. In *Reasoning Web: 4th International Summer School 2008, tutorial lectures* (1-53). Venice, Italy,: Springer-Verlag.

EL BOUHISSI, H., MALKI, M. & BOUCHIHA, D. 2008. A reverse engineering approach for the Web Service Modeling Ontology specifications. In: Proceedings of the Second International Conference on Sensor Technologies and Applications (SENSORCOMM '08) 25-31 August. Cap Esterel

ELENIUS, D., DENKER, G., MARTIN, D., GILHAM, F., KHOURI, J., SADAATI, S., et al. 2005. The OWL-S editor – a development tool for semantic web Services In *The Semantic Web: Research and Applications* (78-92): Springer Berlin / Heidelberg.

ERL, T. 2008. *SOA: principles of service design*: Prentice Hall.

FACCA, F. M., KOMAZEC, S. & TOMA, I. 2009. WSMX 1.0: a further step toward a complete semantic execution environment. In: Proceedings of the 6th Annual European Semantic Web Conference (ESWC2009). 31 May - 4 June. Heraklion, Greece.

FEIER, C., ROMAN, D., POLLERES, A., DOMINGUE, J., STOLLBERG, M. & FENSEL, D. 2005. Towards intelligent Web services: the Web Service Modeling Ontology (WSMO). In: Proceedings of the International Conference on Intelligent Computing (ICIC). 23-26 August. Hefei, China.

FENSEL, D. & BUSSLER, C. 2002. The Web Service Modeling Framework (WSMF). *Electronic Commerce Research and applications,* 1(1).

FIELDING, R. T. 2000. *Architectural Styles and the Design of Network-based Software Architectures.* University of California, Irvine.

FILHO, O. F. F. & FERREIRA, M. A. G. V. 2009. Semantic Web Services: a RESTful approach. In: Proceedings of the IADIS International Conference WWW/Internet 2009 Rome, Italy.

FRIEDMAN-HILL, E. 2003. *Jess in Action: Java Rule-Based Systems*. New York: Manning Publications Co.

GARCÍA-CASTRO, R., YATSKEVICH, M., SANTOS, C. T. D., WRIGLEY, S. N., CABRAL, L., NIXON, L., et al. 2011. The state of semantic technology today – Overview of the First SEALS Evaluation Campaigns. [Online]. Available from: http://www.seals-project.eu/news/902-community/132-seals-whitepaper-semantic-technology. [Accessed: 19 May 2011].

GARCÍA-SANCHEZ, F. 2007. *Knowledge Technologies-Based System for Semantic Web Services Environments.* University of Murcia, Spain.

GARCÍA-SÁNCHEZ, F., SABUCEDO, L. Á., MARTÍNEZ-BÉJAR, R., RIFÓN, L. A., VALENCIA-GARCÍA, R. & GÓMEZ, J. M. 2011. Applying intelligent agents and semantic web services in eGovernment environments. *Expert Systems*, 1-21.

GARCIA-SANCHEZ, F., VALENCIA-GARCIA, R., MARTINEZ-BEJAR, R. & FERNANDEZ-BREIS, J. T. 2009. An ontology, intelligent agent-based framework for the provision of semantic web services. *Expert Systems with Applications,* 36, 3167–3187.

GERBER, A. J., BARNARD, A. & VAN DER MERWE, A. J. 2007. Towards a semantic Web layered architecture. In: Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering. February. Innsbruck, Austria.

GINIGE, A. 2002. Web engineering: managing the complexity of web systems development. In: Proceedings of the 14th international conference on Software engineering and knowledge engineering. 15-19 July. Ischia, Italy.

GOMEZ-PEREZ, A. & EUZENAT, J. (Eds.). 2005. *The Semantic Web: research and applications*: Springer-Verlag Heidelberg.

GOMEZ-PEREZ, A., GONZALEZ-CABERO, R. & LAMA, M. 2004. ODE SWS: a framework for designing and composing semantic Web services. *Intelligent Systems, IEEE,* 19(4), 24-31.

GOTTSCHALK, K., GRAHAM, S., KREGER, H. & SNELL, J. 2002. Introduction to web service architecture. *IBM Systems Journal,* 41(2), 170-177.

GREENWOOD, D. & CALISTI, M. 2004. Engineering Web service - agent integration. In: Proceedings of the IEEE International Conference on Systems, Man and Cybernetics. 10-13 October. The Hague, Netherlands.

GRONMO, R., SKOGAN, D., SOLHEIM, I. & OLDEVIK, J. 2004. Model-driven Web services development. In: Proceedings of the IEEE International Conference on e-Technology, e-Commerce and e-Service. 28-31 March. Hong Kong.

GRUBER, T. R. 1993. A translation approach to portable ontologies. *Knowledge Acquisition,* 5(2), 199-220.

GU, Q. & LAGO, P. 2007. A stakeholder-driven service life cycle model for SOA. In: Proceedings of the 2nd international workshop on service oriented software engineering: in conjunction with the 6th ESEC/FSE joint meeting. Dubrovnik, Croatia.

GUHA, R. 2009. Toward the Intelligent Web Systems. In: Proceedings of the First International Conference on Computational Intelligence, Communication Systems and Networks (CICSYN '09).

GÜMÜS, Ö., GÜRCAN, Ö., KARDAS, G., EKINCI, E. & DIKENELLI, O. 2007. Engineering an MAS Platform for Semantic Service Integration Based on the SWSA. In MEERSMAN, R. & TARI, Z. (Eds.), *Workshops on the move to meaningful Internet Systems (OTM 2007)* (Vol. 4805: 85-94): Springer Berlin / Heidelberg.

GÜRCAN, Ö., KARDAS, G., GÜMÜS, Ö., EKINCI, E. & DIKENELLI, O. 2007. An MAS Infrastructure for Implementing SWSA Based Semantic Services. In HUANG, J., KOWALCZYK, R., MAAMAR, Z., MARTIN, D., MÜLLER, I., STOUTENBURG, S., et al. (Eds.), *Service-Oriented Computing: Agents, Semantics, and Engineering* (Vol. 4504: 118-131): Springer Berlin / Heidelberg.

HADLEY, M. 2009. Web Application Description Language. [Online]. Available from: http://www.w3.org/Submission/wadl/. [Accessed: 14 November 2010].

HASSANZADEH, A., NAMDARIAN, L. & ELAHI, S. B. 2011. Developing a framework for evaluating service oriented architecture governance (SOAG). *Knowledge-Based Systems*.

HEMAYATI, M. S., MOHSENZADEH, M., SEYYEDI, M. A. & YOUSEFIPOUR, A. 2010. A framework for integrating web services and multi-agent systems. In: Proceedings of the 2nd International Conference on Software Technology and Engineering (ICSTE). 3-5 October. San Juan, PR

HENDLER, J. 2001. Agents and the semantic Web. *IEEE Intelligent Systems,* 16(2), 30-37.

HERNANDEZ, R. L. 2007. *A flexible model for the semi-automatic location of services.* Universidad Autonoma de Madrid, Madrid, Spain.

HEROLD, M. 2008. *WSMX Documentation*. Galway, Ireland: Digital Enterprise Research Institute (DERI).

HOFSTEE, E. 2006. *Constructing a good dissertation: a practical guide to finishing a master's, MBA, or PhD on schedule.* Johannesburg, South Africa: EPE.

HORRIDGE, M., JUPP, S., MOULTON, G., RECTOR, A., STEVENS, R. & WROE, C. 2007. *A practical guide to building OWL ontologies using Prot´eg´e 4 and CO-ODE tools*: The University Of Manchester.

HORROCKS, B., PARSIA, P., PATEL-SCHNEIDER, P. & HENDLER, J. 2005. Semantic Web architecture: stack or two towers? In *Principles and Practice of Semantic Web Reasoning* (Vol. 3703/2005: 37-41): Springer Berlin / Heidelberg.

HORROCKS, I. 2008. Ontologies and the semantic web. *Communications of the ACM,* 51(12), 58-67.

HORROCKS, I. & SATTLER, U. 2002. Description Logics -basics, applications, and more. [Online]. Available from: http://www.cs.man.ac.uk/~horrocks/Slides/ecai-handout.pdf. [Accessed: 19 April 2010].

HYVONEN, E. (Ed.). 2002. *Semantic Web kick-off in Finland: vision, technologies, research, and applications*. Helsinki: HIIT Publications, 2002-001, Helsinki Institute for Information Technology (HIIT).

JANEV, V. & VRANES, S. 2010. Applicability assessment of Semantic Web technologies. *Information Processing & Management*.

JENNINGS, K. & WOOLDRIDGE, M. 1996. Software Agents. *IEEE Review*, 17-20.

JENNINGS, N. R. & WOOLDRIDGE, M. 1998. Applications of intelligent agents. In *Agent technology: foundations, applications, and market* (3 - 28): Springer-Verlag New York, Inc.

JOO, J. 2011. Adoption of Semantic Web from the perspective of technology innovation: A grounded theory approach. *International Journal of Human-Computer Studies,* 69(3), 139-154.

JORDAAN, G. D. & LATEGAN, L. O. K. 2010. *Modelling as Research Methodology*: SUN PRESS.

KANELLOPOULOS, D. & KOTSIANTIS, S. 2006. Towards intelligent wireless Web services for tourism. *International Journal of Computer Science and Network Security (IJCNS),* 6(7B), 83-90.

KASHYAP, V., BUSSLER, C. & MORAN, M. 2008. *The Semantic Web: semantics for data and services on the Web*. Berlin: Springer-Verlag.

KEIDL, M. & KEMPER, A. 2004. Towards context-aware adaptable web services. In: Proceedings of the 13th International World Wide Web Conference. 17-22May. New York, USA.

KELLY, P., CODDINGTON, P. & WENDELBORN, A. 2006. A simplified approach to web service development. In: Proceedings of the 2006 Australasian workshops on Grid computing and e-research - Volume 54. Hobart, Tasmania, Australia.

KERRIGAN, M. 2006. WSMOViz: An Ontology Visualization Approach for WSMO. In: Proceedings of the 10th International Conference on Information Visualization.

KERRIGAN, M., MOCAN, A., TANLER, M. & FENSEL, D. 2007. The Web Service Modeling Toolkit (WSMT)- an integrated development environment for Semantic Web Services. In: Proceedings of the 4th European conference on the Semantic Web: Research and Applications. Innsbruck, Austria.

KILIAN-KEHR., R. 2008. *Service engineering and consumption research roadmap* (White paper): SAP Research.

KIRDA, E., JAZAYERI, M., KERER, C. & SCHRANZ, M. 2001. Experiences in engineering flexible Web services. *Multimedia, IEEE,* 8(1), 58-65.

KOIVUNEN, M.-R. & MILLER, E. 2002. *W3C Semantic Web activity*. Helsinki, Finland: HIIT Publications, 2002-001, Helsinki Institute for Information Technology (HIIT).

KONTOGIANNIS, K., LEWIS, G. A. & SMITH, D. B. 2008. A research agenda for service-oriented architecture. In: Proceedings of the 2nd international workshop on Systems development in SOA environments. Leipzig, Germany.

KONTOGIANNIS, K., LEWIS, G. A., SMITH, D. B., LITOIU, M., MULLER, H., SCHUSTER, S., et al. 2007. The Landscape of Service-Oriented Systems: A research perspective. In: Proceedings of the International Workshop on Systems Development in SOA Environments.

KOPECKY, J. & VITVAR, T. 2008. WSMO-Lite: Lowering the Semantic Web Services Barrier with Modular and Light-Weight Annotations. In: Proceedings of the IEEE International Conference on Semantic Computing. 4-7 August Santa Clara, CA

KUROPKA, D., TROGER, P., STAAB, S. & MATHIAS, W. (Eds.). 2008. *Semantic Service Provisioning*.

LARA, R., POLLERES, A., LAUSEN, H., ROMAN, D., DE BRUIJN, J. & FENSEL, D. 2005. *A conceptual comparison between WSMO and OWL-S*: DERI.

LARA, R., ROMAN, D., POLLERES, A. & FENSEL, D. 2004. A conceptual comparison of WSMO and OWL-S. In: Proceedings of the European Conference on Web Services (ECOWS 2004).

LASSILA, O., VAN HARMELEN, F., HORROCKS, I., HENDLER, J. & MCGUINNESS, D. L. 2000. The semantic Web and its languages. *IEEE Intelligent Systems,* 15(6), 67-73.

LAUSEN, J. F. H. 2007. Semantic Annotations for WSDL and XML Schema. [Online]. Available from: http://www.w3.org/TR/sawsdl/. [Accessed: 18 January 2010].

LAUTENBACHER, F. 2006. *A UML profile and transformation rules for semantic web services* (No. 2006-20): Institute of Computer Science, University of Augsburg, Germany.

LEWIS, D. 2008. Intelligent agents and the Semantic Web: developing an intelligent Web. [Online]. Available from: http://www.ibm.com/developerworks/web/library/wa-intelligentage/. [Accessed: 19 June 2009].

LIA, K., ABELA, C. & SCICLUNA, J. 2009. WISE - Workbench for Semantic Web Services. In: Proceedings of the Third International Conference on Advances in Semantic Processing (SEMAPRO '09) 11-16 October. Sliema, Malta.

LOSADA, S., RIBAS, J., CONTRERAS, J., BAS, J. L., BELLIDO, S., GÓMEZ, J. M., et al. 2005. *Mortgage Comparison Service* (No. FP6 - 507483): DIP.

LU, J., ZHANG, G. & RUAN, D. (Eds.). 2007. *E-Service Intelligence: Methodologies, Technologies and Applications* (Vol. 37): Springer-Verlag Berlin Heidelberg.

MARTIN, D., BURSTEIN, M., HOBBS, J., LASSILA, O., MCDERMOTT, D., MCILRAITH, S., et al. 2004. OWL-S: Semantic Markup for Web Services [Online]. Available from: http://www.w3.org/Submission/OWL-S/. [Accessed: 18 June 2009].

MCGUINNESS, D. L. & VAN HARMELEN, F. 2004 OWL Web Ontology Language. [Online]. Available from: http://www.w3.org/TR/owl-features/. [Accessed: 18 November 2009].

MCILRAITH, S., SON, T. & ZENG, H. 2001. Semantic Web Services. *IEEE Intelligent Systems. Special Issue on the Semantic Web,* 16(2), 46 – 53.

MOULIN, C., SBODIO, M. & BETTAHAR, F. 2005. Semantic requirements for eGovernment services interoperability. In: Proceedings of the International Workshop on Semantics and Orchestration of eGovernment Processes. Compiègne, France.

MTSWENI, J., BIERMANN, E. & PRETORIUS, L. 2010. Toward a service creation framework: a case of intelligent semantic services. In: Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists. Bela Bela, South Africa.

NASSAR, M., ANWAR, A., EBERSOLD, S., ELASRI, B., COULETTE, B. & KRIOUILE, A. 2009. Code generation in VUML profile: A model driven approach. In: Proceedings of the IEEE/ACS International Conference on Computer Systems and Applications. 10 -13 May. Rabat, Morocco

NEWCOMER, E. 2004. *Understanding Web Services: XML, WSDL, SOAP, and UDDI*: Pearson.

NWANA, H. S. 1996. Software Agents: an overview. *Knowledge Engineering Review,* 11(3), 1-40.

OBERLE, D., STAAB, S., STUDER, R. & VOLZ, R. 2005. Supporting application development in the semantic web. *ACM Transactions on Internet Technology (TOIT),* 5(2), 328-358.

OBITKO, M. 2007. *Translations between ontologies in multi-agent systems,.* Czech Technical University, Prague.

OLIVIER, M. S. 2006. *Information Technology Research: a practical guide for computer science and informatics* (2nd ed.). Pretoria, South Africa: Van Schaik.

OMG. 2010a. Model Driven Architecture: the architecture of choice for a changing world. [Online]. Available from: http://www.omg.org/mda/executive_overview.htm. [Accessed: 10 February 2011].

OMG. 2010b. OMG Model Driven Architecture. [Online]. Available from: http://www.omg.org/mda/. [Accessed: 10 February 2010].

PAPAZOGLOU, M. P. 2001. Agent-oriented technology in support of e-business. *Communications of the ACM,* 44(4), 71-77.

PAPAZOGLOU, M. P. & VAN DEN HEUVEL, W. J. 2006. Service-oriented design and development methodology. *International Journal of Web Engineering and Technology,* 2(4), 412 - 442.

PASCHKE, A. & BICHLER, M. 2008. Knowledge representation concepts for automated SLA management. *Decision Support Systems,* 46(1), 187-205.

PAUTASSO, C., ZIMMERMANN, O. & LEYMANN, F. 2008. RESTful Web Services vs. “Big” Web Services: making the right architectural decision. In: Proceedings of the WWW 2008. 21-25 April. Beijing, China.

PROTOGEROS, N. 2008. *Agent and Web services technologies in Virtual Enterprises*.

QAFMOLLA, X. & CUONG, N. V. 2010. Automation of Web services development using model driven techniques. In: Proceedings of the 2nd International Conference on Computer and Automation Engineering (ICCAE). 26-28 February. Singapore

REBSTOCK, M. 2009. Technical opinion: semantic ambiguity. *Communications of the ACM,* 52(5), 145-146.

RICHARDSON, L. & RUBY, S. 2007. *RESTful Web Services*. Sebastopol, USA: O’Reilly Media.

RINGELSTEIN, C., FRANZ, T. & STAAB, S. 2007. The Process of Semantic Annotation of Web Services. In CARDOSO, J. (Ed.), *Semantic Web Services: theory, tools, and applications* (217-239). Germany: IGI Global.

RIVIÈRES, J. D. & WIEGAND, J. 2004. Eclipse: a platform for integrating development tools. *IBM Systems Journal,* 43(2), 371-383.

ROMAN, D., DE BRUIJN, J., MOCAN, A., LAUSEN, H., DOMINGUE, J., BUSSLER, C., et al. 2006. WWW: WSMO, WSML, and WSMX in a nutshell. In: Proceedings of the 1st Asian Semantic Web Conference 3-7 September. Beijing, China.

SABOU, M., WROE, C., GOBLE, C. & STUCKENSCHMIDT, H. 2005. Learning domain ontologies for semantic Web service descriptions. *Web Semantics: Science, Services and Agents on the World Wide Web,* 3(4), 340-365.

SASSEN, A. & MACMILLAN, C. 2005. *The service engineering area: an overview of its current state and a vision of its future*. Belgium: EUROPEAN COMMISSION (EC).

SCHAAF, M. & MAURER, F. 2001. Integrating Java and CORBA: a programmer's perspective. *IEEE Internet Computing,* 5(1), 72-78.

SHEN, W., LI, Y., QI, H. W., S. & GHENNIWA, H. 2005. Implementing collaborative manufacturing with intelligent Web services. In: Proceedings of the Fifth International Conference on Computer and Information Technology (CIT'05)

SHENG, Q. Z., YU, J., SEGEV, A. & LIAO, K. 2010. Techniques on developing context-aware web services. *International Journal of Web Information Systems,* 6(3), 185-202.

SIMULA, K. 2007. Intelligent software agent framework for customized mobile services. In: Proceedings of the 4th Middleware Doctoral Symposium. Newport Beach, California.

SIORPAES, K. & SIMPERL, E. 2010. Human Intelligence in the Process of Semantic Content Creation. *World Wide Web (WWW) Journal,* 13(1-2), 33-59.

SMITH, M. K., WELTY, C. & MCGUINNESS, D. L. 2004. OWL Web Ontology Language guide. [Online]. Available from: http://www.w3.org/TR/2004/REC-owl-guide-20040210/#OwlVarieties. [Accessed: 03 October 2009].

SOE-TSYR, Y. & KWEI-JAY, L. 2003. WISE-building simple intelligence into Web services. In: Proceedings of the IEEE/WIC International Conference on Web Intelligence (ICWI03). 13-17 October. Halifax, Canada.

SOMMERVILLE, I. 2006. *Software Engineering: update* (8th ed.). Amsterdam, Netherlands: Addison-Wesley

SRINIVASAN, N., PAOLUCCI, M. & SYCARA, K. 2005. *CODE: a development environment for OWL-S Web services.* (No. Technical Report CMU-RI-TR-05-48): Robotics Institute, Carnegie Mellon University.

SRINIVASAN, N., PAOLUCCI, M. & SYCARA, K. 2006. Semantic Web Service Discovery in the OWL-S IDE. In: Proceedings of the 39th Annual Hawaii International Conference on System Sciences 4-7 January. Kauai, Hawaii, USA.

STAHL, T. & VOLTER, M. 2006. *Model-driven software development: technology, engineering, management*: John Wiley & Sons, Ltd.

STEIN., S. 2008. *Flexible service provisioning in multi-agent systems.* Unpublished Thesis, University of Southampton.

STOJANOVIC, Z. & DAHANAYAKE, A. (Eds.). 2005. *Service-Oriented Software System Engineering: challenges and practices*. Singapore: Idea Group Publishing.

STOLLBERG, M. 2006. State of Affairs in Semantic Web Services. *Canadian Semantic Web Symposium* [Online]. Available from: http://www.wsmo.org/TR/d17/resources/200606-CSWWS2006/SWStutorial-CSWWS2006.pdf. [Accessed: 15 September 2010].

STOLLBERG, M., HEPP, M. & FENSEL, D. 2010. Semantics for Service-Oriented Architectures. In GRIFFITHS, N. & CHAO, K.-M. (Eds.), *Agent-Based Service-Oriented Computing,* : Springer.

STOLLBERG, M., LAUSEN, H., ARROYO, S., HERZOG, R., SMOLLE, P. & FENSEL, D. 2004. *FRED whitepaper: an agent platform for the semantic Web*. Ireland: DERI-Digital Enterprise Research Institute.

STUDER, R., GRIMM, S. & ABECKER, A. (Eds.). 2007. *Semantic web services: concepts, technologies, and applications*: Springer-Verlag Berlin.

SUH, W. 2005. *Web engineering: principles and techniques*. Retrieved 04 November 2010, from http://books.google.com/books?id=MPQhKGjl6tUC&pg=PA58&source=gbs_selected_pages&cad=3#v=onepage&q&f=false.

SUN MICROSYSTEMS, I. 2009a. The Java API for XML-Based Web Services (JAX-WS) 2.2: Sun Microsystems, Inc.

SUN MICROSYSTEMS, I. 2009b. JAX-RS: Java™ API for RESTful Web Services.

SUN, W., LI, S., ZHANG, D. & YAN, Y. 2009. A Model-Driven Reverse Engineering Approach for Semantic Web Services Composition. In: Proceedings of the World Congress on Software Engineering (WCSE '09). Xiamen

SUN, Z. & LAU, S. K. 2007. Customer experience management in e-services *Studies in Computational Intelligence (SCI),* 37, 365-388.

THO, Q. T., FONG, A. C. M. & HUI, S. C. 2007. A scholarly semantic web system for advanced search functions. *Online Information Review,* 31(3), 353-364.

TOCH, E., REINHARTZ-BERGER, I. & DORI, D. 2011. Humans, semantic services and similarity: A user study of semantic Web services matching and composition. *Web Semantics: Science, Services and Agents on the World Wide Web,* 9(1), 16-28.

TORRES, V., PELECHANO, V. & PASTOR, Ó. 2006. Building Semantic Web Services based on a model driven Web engineering method In *Advances in Conceptual Modeling - Theory and Practice* (Vol. 4231/2006: 173-182): Springer Berlin / Heidelberg.

TSAI, W. T. 2005. Service-oriented system engineering: a new paradigm. In: Proceedings of the IEEE International Workshop on Service-Oriented System Engineering. 20-21 October. Beijing, China.

USMAN, A. M., NADEEM, M., ANSARI, Z. A. & RAZA, S. 2006. Multi-agent Based Semantic E-government Web Service Architecture Using Extended WSDL. In: Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology Workshops. Hong Kong

VAN DEN HEUVEL, W. J., ZIMMERMANN, O., LEYMANN, F., LAGO, P., SCHIEFERDECKER, I., ZDUN, U., et al. 2009. Software service engineering: tenets and challenges. In: Proceedings of the ICSE Workshop on principles of engineering service oriented systems,.

VITVAR, T., KOPECKY, J. & FENSEL, D. 2009. WSMO-Lite: a lightweight semantic descriptions for services on the Web. [Online]. Available from: http://cms-wg.sti2.org/TR/d11/v0.2/20090310/. [Accessed: 12 August 2010].

W3C. 2005. W3C Extensible Markup Language (XML). [Online]. Available from: http://www.w3.org/XML/. [Accessed: 20 October 2009].

WAHID, F. 2007. Using the technology adoption model to analyze Internet adoption and use among men and women in Indonesia. *The Electronic Journal on Information Systems in Developing Countries,* 32(6), 1-8.

WANG, X., KRÄMER, B. J., ZHAO, Y. & HALANG, W. A. 2007. Representation and discovery of intelligent e-services. *Studies in Computational Intelligence (SCI),* 37, 233-252.

WEBODE. 2003. WebODE ontology engineering platform. [Online]. Available from: http://webode.dia.fi.upm.es/WebODEWeb/index.html. [Accessed: 2 April 2009].

WHITE, S. A. 2004. *Introduction to BPMN*. IBM Corporation,

WRIGLEY, S. N., REINHARD, D., ELBEDWEIHY, K., BERNSTEIN, A. & CIRAVEGNA, F. 2011. Methodology and campaign design for the evaluation of semantic search tools. In: Proceedings of the 3rd International Semantic Search Workshop. Raleigh, North Carolina.

XIAOFENG, Y., JUN, H., YAN, Z., TIAN, Z., LINZHANG, W., JIANHUA, Z., et al. 2006. A Model Driven Development Framework for Enterprise Web Services. In:

Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference. October. Hong Kong.

YE, R. & YANG, X. 2009. Multi-Agent Web Services Aggregation Driven by Requirement in JADE. In: Proceedings of the International Symposium on Computer Network and Multimedia Technology. Wuhan

YEGANEH, S. H., HABIBI, J., ROSTAMI, H. & ABOLHASSANI, H. 2010. Semantic web service composition testbed. *Computers & Electrical Engineering,* 36(5), 805-817.

YU, L. 2007. *Introduction to the Semantic Web and Semantic Web Services*. New York: Chapman & Hall/CRC.

YU, W. D. & ONG, C. H. 2009. A SOA-based software engineering design approach in service engineering. In: Proceedings of the IEEE International Conference one-Business Engineering.

ZHANG, L., ZHANG, J. & CAI, H. 2007. *Services Computing*: Springer Heidelberg.

ZHU, H. & SHAN, L. 2005. Agent-oriented modelling and specification of Web services. In: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. 2-4 February. Sedona, Arizona.