Model Driven Service Description and Discovery Framework for Carrier Applications

by

Nikolaos Giannopoulos

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Sciences
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The most dominant architecture in the contemporary business domain is Service Oriented Architecture (SOA). The large number of the existing service description and discovery systems available today, including the ones proposed in research proposals, reveals an increasing need for adaptive, semantically enriched and context-aware, wide-area service discovery. This need will become more intense in the years to come as the number of available services increases rapidly. The main reason behind the existence of a plethora of such systems is that before these initiatives, the standard in service discovery was taking into account only the syntactic descriptions of the services, causing conflicts when services, with similar syntactic descriptions, needed to be evaluated.

The research solutions available today offer efficient and accurate discovery at the syntactic, functional semantic and non-functional semantic level. However, the problem is that there is no general consensus yet regarding service discovery. Research by its very nature, leads to point solutions rather than complete systems.

Based on these observations, we propose an adaptive service description and discovery framework for carrier applications, enabling the model-driven specification of services and client profiles, and also, for allowing the dynamic configuration of the services to meet specific quality requirements defined by the clients. The framework was implemented in the context of Model Driven Development, to ensure platform independence at the level of the specification of services. The framework takes the union of the point solutions offered by research proposals in the area of service description and discovery, creates an abstract model, and can compile that model to platform specific code. More specifically, services for carrier applications can be specified in a platform independent way both in terms of

service signatures (syntactic properties) and in terms of the functionality and the QoS service characteristics (semantic properties). A model transformation framework allows for the creation of a platform specific model for the description of services in a specific technology platform (*e.g.*, Web services). The framework is extensible to accommodate future extensions. In addition, as a proof of concept, we designed and developed an Eclipse Rich Client Platform (RCP) prototype tool, implementing our proposal.

Acknowledgements

My deepest gratitude goes to my supervisor, Professor Kostas Kontogiannis, for his guidance, advice and trust he showed towards me all this time. Without him this thesis would have never been realized. I was truly honored to be one of his graduate students and to be able to collaborate with him for the past two years. Most importantly I am grateful that I got to know not only a great person but also a good friend. The experiences and knowledge I gained during this period cannot be measured in any way and I feel privileged having the opportunity to experience everything. I know for a fact that this was the most important and exciting journey of my life until now.

I would also like to thank the readers of my thesis Professors Paul A.S. Ward and Paul P. Dasiewicz, for their valuable contribution, comments and suggestions.

Words cannot describe my appreciation and love towards my family that supported me in this journey and were always by my side.

Finally my deepest and sincere thanks go to Susan. Without her I would have never made it thus far. She was the one that gave my stay in Canada a special and priceless meaning that will always be in my heart and memories.

Dedication

To my beloved father and mother, Kostas Giannopoulos and Eleni Giannopoulou. I owe everything to you.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation - Problem Description

Services can be defined as entities that add value from enhancing the capabilities of things (such as customizing, distributing, *etc.*) and interactions between things. The interpretation of what a service is varies according to the application domain. In the telecom industry, the connectivity is considered to be a service, in the banking industry, a bank account is considered to be a service, in the computing field a service is a well-defined API and so on. Services, within the context of the current thesis, are software components with a well-defined interface (API) and properties used to describe, discover and invoke the service. The API of such a service can be accessed over a network protocol.

A technology-specific specialization of a service is a Web service. The World Wide Web Consortium (W3C)[1] defines a Web service as "a software system designed to support interoperable Machine to Machine interaction over a network". Web services are frequently

[1]http://www.w3.org/

just Web APIs that can be accessed over a network, such as the Internet, and executed on a remote system hosting the requested services[2]. In the consumer domain, people are interacting with a variety of heterogeneous devices, such as desktop computers, laptops (*e.g.*, e-mailing, instant messaging, video conferencing, file transferring), ATMs, *etc.* In addition, the emergence of mobile devices, such as cell phones and palmtops, facilitates key everyday activities of a person [32]. All of these devices have a common characteristic, that is to provide the desired functionalities to their users, by utilizing a number of external services including, in some cases, Web services. Other examples of Web services are Google[3] and Amazon[4] that utilize Web services in order to process client requests on their websites.

In the business domain, the use of service-oriented systems is rapidly replacing the existing traditional approaches. In particular, a service centric architectural style known as Service Oriented Architecture (SOA) is the most prominent one. SOA offers a number of important qualities. It aligns business people with IT people to use the same or similar terminology. It is the current trend, allowing business people to visualize software under a different perspective. Software can be treated as higher level services providing abstractions of what a software component is, facilitating business people in the understanding of software components. SOA involves three major activities: service provision, service discovery and service consumption. Service discovery, in the context of SOA, has a number of interesting opportunities such as combining services in workflows, introducing and using user profiles, introducing a more efficient context of service invocation, and allowing future enhancements in utilizing service discovery.

However, the core problems of the traditional approaches even after the introduction

[2] http://en.wikipedia.org/wiki/Web_service
[3] http://www.google.com/
[4] http://www.amazon.com/

of the service-oriented systems are unaltered. The underlying technology may change, but the basic issues remain. There is a plethora of research proposals addressing the issue of service description and service discovery. Examples of such proposals are DHCP[5], UPnP[6], SLP[7], UDDI[8], X.500[9], *etc.* There is significant work on non-functional semantic discovery and selection included in [15], [37] and [14]. Although each of the existing proposals contributes significantly in the area of service description and discovery, there is not yet an approach that can be considered a strong candidate to become a standard in the near future. That is the reason we chose to implement our framework in the context of Model Driven Development (MDD), to enable us to abstract the common aspects of existing techniques in an abstract model that can be re-deployed according to the requirements of a business domain, so business doesn't need to keep changing its code each time the technology changes.

MDD is the current trend in defining new frameworks and techniques or implementing new software, because it offers a number of important qualities [59]. First, it is advocated that it reduces the cost of software development by generating code and artifacts from models, increasing developer productivity. Second, high-level models are kept free of irrelevant implementation detail, making it easier to handle changes in the underlying platform technology and its technical architecture, thus improving maintainability. Third, it aims to improve reusability, adaptability and consistency. Fourth, it reduces the risk of error prone code when thoroughly tried and tested transformations are repeatedly re-used. Fifth, it improves the stakeholders' communication, because models are easier to comprehend than

[5]http://en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol
[6]http://en.wikipedia.org/wiki/Universal_Plug_and_Play
[7]http://en.wikipedia.org/wiki/Service_Location_Protocol
[8]http://en.wikipedia.org/wiki/Universal_Description_Discovery_and_Integration
[9]http://en.wikipedia.org/wiki/X.500

technology specific artifacts. Sixth, it improves the design communication and allows the design models to be always synchronized with the implementation. Seventh, it captures the expertise of the developers more efficiently. Eight, models become important long-term assets of each organization. Finally the developers may start the development process before the targeted platforms have been decided, accelerating the development process.

The model-driven development of service descriptions is the novelty of the current thesis.

1.2 Thesis Contributions

This thesis aims to address the aforementioned problems by proposing a model-driven service description and discovery framework for carrier applications, consisting of the necessary tools and techniques. The major contributions of this thesis are:

1. The design of a syntactic-based service interface description specification; the specification defines the syntactic details (*e.g.*, the operations of the service, the parameters of the operations, *etc.*) of a service in a service-oriented environment;

2. The design of a semantic-based service description specification; the specification defines the semantic details (*e.g.*, preconditions, quality of service, management policies, *etc.*) of a service in a service-oriented environment;

3. The design of a Platform Independent Model (PIM) for service-oriented systems containing the aforementioned syntactic and semantic specifications; the PIM will provide the means for a thorough categorization of services based on both their functional and non-functional characteristics;

4. The design of a Platform Specific Model (PSM) for Web Services accommodating the specifications defined in the PIM in a platform-specific manner; the PSM essentially corresponds to an extended WSDL 1.1 metamodel; the extensions contain the semantic-specific definitions of the metamodel that are not included in the WSDL 1.1 specification;

5. The design and implementation of a transformation model, in the context of OMG's Meta-Object Facility (MOF)[10], describing the transformations and mappings between the PIM model and the PSM model; the transformation model was implemented using the ATL model transformation language;

6. A second transformer, transforming the generated PSM models into WSDL 1.1 source code; keep in mind that the code will conform to an extended version of the WSDL 1.1 specification because it will include all the semantic information specified for the service;

7. The design of a PIM model for the definition of a client's profile including the client's location, his/her personal information and preferences;

8. The definition of an adaptive service selection framework; the adaptiveness is added with the introduction of the semantic information in both the service and the client descriptions, and, additionally, with the introduction of different policies defining the service selection process (the algorithm); these policies will be implemented using the Factory design pattern, allowing the effortless addition of new policies as necessary; the framework allows the dynamic configuration of the Web Services to meet specific

[10] http://www.omg.org/mof/

quality requirements defined by the clients;

1.3 Thesis Layout

The rest of the thesis is organized as follows. Chapter 2 provides a survey of existing research in areas related to the work presented in this thesis, which includes a discussion on Model Driven Development (MDD), Service Oriented Architecture (SOA), Semantic Web, Semantic Web realization initiatives (*e.g.*, Ontologies, WSMF *etc.*), semantic Web Service selection and model transformations. Chapter 3 presents the Platform Independent Model (PIM) for service-oriented systems (PIM Lite), specified for the purposes of this thesis. The description of the full version of our PIM is contained in Appendix A. Chapter 4 provides the description of the semantic service description framework, used to define semantic specific information for both the services and their clients. Chapter 5 describes the Platform Specific Model (PSM) for Web Services in carrier applications, specified for the purposes of this thesis. Chapter 6 discusses the model transformation framework that includes two phases: the first phase includes the transformation from PIM to PSM and the second phase includes the transformation from PSM to source code. Chapter 7 presents the service selection framework. Chapter 8 provides a case study implementing everything we propose, as a proof of concept. Finally, Chapter 9 presents the conclusions and discusses avenues for future work.

Chapter 2

Related Work

2.1 Model Driven Development

One of the major problems, the software engineering community faces, is software complexity. As stated in [48], this inherent complexity is enhanced from the contemporary competitive market pressures and the continuous pursuit of greater productivity, under the pressure of tight delivery deadlines. As the demands of the modern society and industry grow every day, the complexity of the software will increase with them.

Everyone who has programming experience is aware of how demanding it is to write code for complex applications. It is even more demanding and challenging to understand code written by another programmer and sometimes it is even difficult to understand code written by you some time ago. In addition, large industrial software programs need to be fault tolerant, meaning that they should be able to continue their operation in the presence of programming faults. Being fault intolerant would mean that the simplest error inside the code could cause unpredictable effects in the behavior of the program itself.

When a software program becomes complex, a programmer would turn to abstracting some components in the code in order to make it more readable and more easily maintainable. This is where MDD comes into play facilitating this task.

Model driven development was introduced as a solution to the aforementioned problems. Initially, the software engineering community questioned the practical value and the potential of software modeling. It was considered just another way to introduce and communicate high-level design ideas and that its only use would have been in the early stages of the development [18]. Once these stages would have passed the models would act as documentation without any further value. During the last few years however, this trend has changed and MDD has been exploited more efficiently and it is rapidly maturing.

The essential artifacts in model driven development are the models. They play a crucial role in the software development process. MDD provides the framework to transform high-level software models to other high-level or lower-level software models. In practice, this is used to transform high-level Platform Independent Models (PIMs) to other high-level Platform Specific Models (PSMs). Moreover, these PSMs can then be transformed to source code, as source code itself can be considered a certain type of a model. The weight in this case falls to the transformers that handle the transformations between the models. An obvious advantage of this approach, assuming the transformers are implemented correctly, is that all the models, participating in the transformations, and the source code are in complete synchronization. Additionally, the programming process is radically accelerated and effectively assisted. The most important and dominant initiative supporting MDD is UML 2[1]. An engineering model of some system is an abstraction of that system that highlights some of its properties from a specific viewpoint. As stated in [48], an engineering

[1] http://www.uml.org/

model should satisfy at least four requirements: it must hide all irrelevant information; it must be easily understandable; it must be accurate; finally, it must allow anyone studying it to predict the behavior of the system.

What makes MDD unique is that it raises the level of abstraction of the specifications, as it brings the specifications closer to the problem domain and further away from the implementation domain, it raises the level of automation, allowing the transformation from high-level model constructs to source code, and, finally, it increases the product quality and the productivity of development. These were the reasons we chose to adopt MDD in our framework.

2.1.1 Model Transformations

Model transformations are essential in Model Driven Development. There are two major model transformation types, namely model-to-model and model-to-text transformations. However, since text can be considered a type of model, both categories, essentially, fall under the model-to-model category. A model is an abstraction of a system or its environment, or both. In software engineering, the term model is often used to refer to abstractions above program code, such as requirements and design specifications. Due to the fact that models are an abstraction mechanism, there is a wide range of software development artifacts as potential transformation models, such as UML models, interface specifications, data schemas, component descriptors, and program code. A very good graphical notation of what is included in a model transformation is provided in [10].

Figure 2.1 provides an overview of the main concepts involved in model transformations. In the example, a simple transformation engine reads an input model conforming to a source

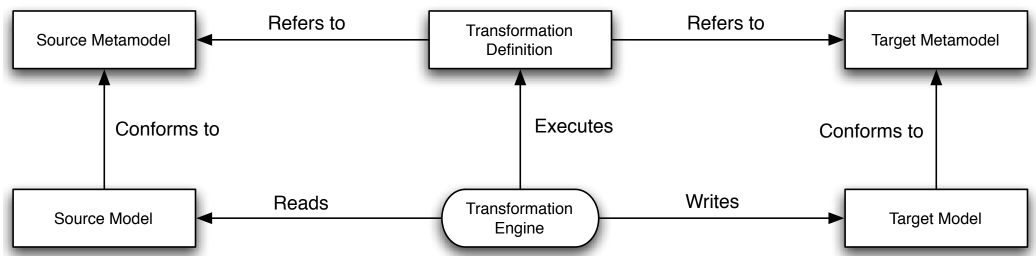


Figure 2.1: Basic concepts of model transformation

metamodel, and writes a target model conforming to a target metamodel, according to the transformation rules defined in the transformation definition module. A transformation is defined with respect to the metamodels. The transformation definitions, executed by the transformation engine, contain rules specifying the transformations (mappings) from the elements of the source metamodel to the elements of the target metamodel. In general, a transformation may have multiple source and target models.

Some of the applications of model transformations include [10]: generating lower-level models, and eventually code, from higher-level models; mapping and synchronizing among models at the same level or different levels of abstraction; creating query-based views of a system; model evolution tasks such as model refactoring; and finally, reverse engineering of higher-level models from lower-level models or code. The current standard for model transformations is QVT (Queries/Views/Transformations) [23] defined by the Object Management Group (OMG)[2]. Although there have been many approaches to model transformations over the last three years, there are no industrial-strength and mature model-to-model transformation systems available. A very thorough and extensive survey is presented in [10]. There is a wide variety of existing model transformation approaches.

[2]http://www.omg.org/

The most important ones are the following:

- VIATRA (VIsual Automated model TRAnsformations) framework [58]. It is the core of a transformation-based verification and validation environment for improving the quality of systems, designed using the UML, by automatically checking consistency, completeness, and dependability requirements.

- Kent Model Transformation language [3]. This language aims to be a declarative specification language, with the option to provide constructive parts if required. The semantics of the language are a combination of relations (OCL relations) and terms from OMG's QVT. It provides a Model Driven Development Environment in which Models and Transformers may be manipulated as first class entities.

- ATL (Atlas Transformation Language) [30]. A simple but powerful Java-like transformation language. A very popular approach. This is the language we used to describe our model-to-model transformations.

- Kermeta [41]. Transforms a MOF-compliant source model (conforming to a MOF metamodel) to a MOF-compliant target model (conforming to a MOF metamodel). The transformations are specified with a transformation model.

- The Core, Relations, and Operational languages, described in the final adopted QVT specification [23]. QVT (Queries/Views/Transformations) is a standard for model transformations defined by the Object Management Group. The QVT standard only addresses model-to-model transformations. The models must conform to any MOF 2.0 metamodel. All transformations of type model-to-text or text-to-model are presently outside the scope of QVT.

- Andro-MDA [1]. An extensible generator framework that adheres to the Model Driven Architecture (MDA) paradigm. It enables models from UML tools to be transformed into deployable components for various platforms (*e.g.*, J2EE, Spring, .NET *etc.*). AndroMDA comes with a host of ready-made cartridges that target today's development toolkits like Axis, jBPM, Struts, JSF, Spring and Hibernate. It also contains a toolkit for building your own cartridges or customize existing ones.

- openArchitectureWare (oAW) [2]. A modular MDA/MDD generator framework implemented in Java. It supports parsing of arbitrary models, and a language family to check and transform models as well as generate code based on them. It has strong support for EMF (Eclipse Modeling Framework) based models but can work with other models, too (*e.g.*, UML2, XML or simple JavaBeans). At the core there is a workflow engine allowing the definition of generator/transformation workflows. A number of pre-built workflow components can be used for reading and instantiating models, checking them for constraint violations, transforming them into other models and then finally, for generating code.

- Fujaba (From UML to Java And Back Again) [43]. It supports the generation of Java source code from UML models, producing an executable prototype. In addition, it offers reverse engineering (to some extend so far, not for productive use), so that Java source code can be parsed and represented within UML.

- JET (Java Emitter Templates) [44]. A powerful tool for generating source code. It is part of Eclipse Modeling Framework (EMF) project. With JET you can use a JSP-like syntax (actually a subset of the JSP syntax) that makes it easy to write templates that express the code you want to generate. JET is a generic template engine that can be used to generate SQL, XML, Java source code and other output from templates. Another popular approach. This approach was used in our model-to-text transformations.

Since it is not our purpose to provide a survey of the existing techniques, for more information please refer to the survey paper [10].

2.2 Service Oriented Architecture

Service Oriented Architecture[3] is an architectural style that gains great momentum inside the software engineering community. SOA offers a number of desirable properties in the business domain [31], as it offers loosely coupled, business-oriented, networked services, which enable flexibility and interoperability. We have to note that these properties where offered in the past by technologies such as CORBA[4] as well.

Although, SOA predates Web Services, currently SOA and Web Services are linked together. An SOA architecture may contain Web Services, which are well defined and independent of the state of other Web Services. These services can communicate with each other, passing messages, using various communication protocols such as SOAP[5]. The desired functionalities are either offered by standalone services or groups of services that collaborate with each other. However, SOA does not depend on Web Services. In addition to the services, in SOA, there are the service consumers. Each Web Service provides a description of its functionalities (usually using WSDL), stored in a repository, and the

[3] http://en.wikipedia.org/wiki/Service-oriented_architecture
[4] http://www.corba.org/
[5] http://www.w3.org/TR/soap/

candidate consumers of these Web Services chose the appropriate Web Services according to their needs. SOA is currently considered a very modular and well-defined architecture.

2.3 Semantic Web

In the context of the current thesis, the technologies composing the traditional Web Services Technology Stack are the following [19]: Simple Object Access Protocol (SOAP), Web Service Description Language (WSDL) and Universal, Description, Discovery and Integration (UDDI). SOAP enables the communication with a Web Service. WSDL describes the Web Service interface[6]. UDDI is a repository enabling publishing and discovering of Web Service specifications and capabilities[7]. However, none of these technologies offers formal semantic descriptions causing the tasks of discovering, selecting, composing, and binding of Web Services to require manual human intervention. In addition, none of these technologies is mandatory.

Tim Berners-Lee introduced the concept of Semantic Web in 2001. The vision of Semantic Web was to act as a complement for the current Web, which is mostly understandable by humans [56]. The intention was to extend the web information in a way to be interpretable and recognizable by machines. By adding semantics in the web content in a standardized way, the machines would be able to understand the content and use it as needed. The application range of Semantic Web is vast, and as a result it has attracted a lot of research and studies around this field. The standard fundamental building blocks of semantic-based systems are the domain specific ontologies. The current standard for

[6]http://www.w3.org/TR/wsdl
[7]http://www.uddi.org/

specifying ontologies is the Web Ontology Language (OWL)[8], issued by World Wide Web Consortium (W3C)[9].

In a service-oriented environment, the architecture may include Web Services. The current standard for describing Web Services is the Web Services Description Language (WSDL), issued also by W3C. However, WSDL can be used only for the syntactic description of a Web Service. Missing semantic information creates a bottleneck in automating the discovery, invocation, composition and contracting of Web Services. Semantic Web can play a crucial role in automating these activities. By using ontologies or other means, we are able to add semantics to our Web Services and as a result take a step forward automating the discovery, invocation, composition, and contracting of Web Services. With this way, we advance from Web Services to Semantic Web Services. In the next section we will discuss why we didn't eventually use ontologies in our framework, but instead we chose to extend the WSDL to support semantic-specific information.

Unfortunately, semantic Web Services do not currently receive the necessary attention from the software engineering community as stated in [27]. It is strongly suggested that the functionalities offered by the Web Services should be semantically annotated, in addition to adding semantics to the data created or consumed by the Web Services. Even if the Web content was somehow perfectly annotated it would have been insufficient to enable the Semantic Web vision, not only by the lack of machine access to Web content, but rather the lack of content itself and additionally the dependence of the content to the state of the business offering the content. Although ontologies seem to be a fairly efficient approach towards specifying Semantic Web Services, in the next section we will explain why we chose

[8]http://www.w3.org/TR/owl-features/
[9]http://www.w3.org/

a different approach.

2.4 Semantic Web Realization Initiatives

In this section we will present the most important contemporary initiatives to annotate semantically the web related content. We must clarify in advance that none of these approaches is a standard until the time of this writing. In this respect, we propose an approach that is an amalgamation of the most prominent specification protocols to semantically annotate Web content. Our intention is not to provide an exhaustive list of Web Service-related semantic properties but to provide an initial list of properties and most importantly provide the framework and the foundations to enable the extension of our approach to accommodate additional semantics that another researcher may come up with. This is the reason why the proposed framework is fully extensible. There are a number of available proposals in an effort to establish an efficient approach to add semantics. One way of adding semantic-specific information is through the use of ontologies. There are three approaches in this direction, namely OWL-S, WSMF and WSMO. Another approach would be to extend WSDL with semantic-specific information. There are also two approaches in this area, namely WSDL-S and SAWSDL. We will present these approaches in the following sub-sections.

2.4.1 Ontologies

The term "ontology" has been thoroughly discussed and explained in nowadays, thus we will provide a brief definition of the term based on the definition provided in [25]. Ontology is defined as an explicit specification of a conceptualization. The term originates from philosophy, and it is used to describe anything that has existence and can be represented. An ontology contains entities and relationships associating different entities. The associated entities may belong to the ontology that is been specified, or may belong to different ontologies. Additionally, there exist formal definitions describing the entities, assigning specific meanings to each entity, as well as formal axioms constraining the interpretation and defining the well-formed use of the terms specified inside the ontology. Ontologies can be considered as well-defined vocabularies used in the representation and sharing of knowledge. The current standard for the definition of ontologies is Web Ontology Language (OWL).

2.4.2 OWL-S

OWL-S [36] is a Web Service ontology, which enables Web Service providers to specify their Web Services both syntactically and semantically, by supplying them with a core set of markup language constructs. Using OWL-S, the Web Service descriptions become interpretable by the computers, enabling the automation of Web Service tasks such as, Web Service discovery, execution, composition and interoperation. OWL-S extends Web Ontology Language (OWL), which is the current standard for specifying ontologies. It is considered to be the most widely accepted and adopted language for specifying semantic Web Service descriptions, it is a well-researched approach, and it is adopted by numerous researchers in both academia and industry. It seems to be a very good candidate, if not the best one, to become a standard in the future. In a nutshell, OWL-S contains three sub-ontologies: the service profile ontology, presenting "what the service does", the abstract definition of the service; the service model ontology describing "how the service works", the concrete definition of the service; finally, the service grounding ontology, presenting "how the service is accessed". The service profile ontology can participate in service advertising, constructing service requests and matchmaking, the service model may participate in service invocation, enactment, composition, monitoring and recovery, whereas the service grounding ontology associates the service with specific communication protocols such as SOAP, providing all the necessary information. There is an overlap between OWL-S and WSDL and thus these two specifications can be combined in a way that would produce a more detailed and accurate description of a Web Service.

2.4.3 Web Service Modeling Framework

Web Service Modeling Framework (WSMF) [17] is a modeling framework for describing Web Services. It includes two basic principles: strong decoupling of its components and the presence of a strong mediation service facilitating the communication between its components. WSMF targets mainly e-commerce applications. WSMF consists of four elements: ontologies, that provide the terminology used by the rest of the components; goal repositories, defining the issues that should be addressed by the Web Services; Web Service descriptions; and finally, mediators, that bypass interoperability problems, such as mediation of data structures when a Web Service provides an input for a second Web Service, however, not in the right format. Since OWL-S seems to be dominating in this area we will not further elaborate on the description of WSMF. For more information you can consult [17].

The Web Service Modeling Ontology (WSMO) [12] is based on WSMF and it is an ontology used in describing semantic Web Services. The ontology refines and extends

WSMF, providing ontological specifications for the core elements of semantic Web Services.

2.4.4 WSDL-S

WSDL-S [4] is currently at a proposal stage, issued by the University of Georgia, aiming to add semantic expressiveness in Web Service descriptions, which is essential to represent the requirements and capabilities of Web Services. It acknowledges the importance of adding semantics in the Web Service descriptions, because these semantics would improve software reusability and discovery, facilitate the composition of Web Services and enable the integration of legacy applications as part of business process integration. It recognizes the lack of semantic expressiveness in WSDL, due to the fact that WSDL offers only syntactic expressiveness, and proposes the exploitation of the extensibility of WSDL, by introducing extensibility elements in various parts of a WSDL document. They assume that formal semantic models for Web Services already exist and they propose to reference these external definitions from inside the WSDL document using extensibility elements. They add semantic information in the inputs, outputs and operations of a Web Service. They additionally introduce the concepts of preconditions and effects for operations, Web Service categorization, and finally, two attributes, “modelReference” and “schemaMapping”. The attribute “modelReference” is used to specify the association between a WSDL entity and a concept in some semantic model. It can be added to a complex type, element, operation, as well as to the extension elements (precondition and effect). The attribute “schemaMapping” is added to XSD elements and complex types, for handling structural differences between the schema elements of a Web Service and their corresponding semantic model concepts.

They argue that their approach is better than OWL-S for two reasons: firstly, the users will be able to specify both syntactic and semantic information from inside the WSDL document, which is an advantage since the developer community is already familiar with WSDL; secondly, their approach is agnostic to ontology representation languages, enabling the users to chose the language they prefer, which could be, for example, UML and not OWL, unlike OWL-S that imposes OWL as the ontology language. Although this approach is quite similar to ours, it differs in the way it introduces new semantics. Our approach is totally ontology agnostic, and it allows the users to extend our schema that defines the semantic properties of the Web Services according to their needs. Nevertheless, these properties could be as well specified inside an ontology. However, the advantages of the WSDL-S language are definitely accurate and valid for our approach as well.

2.4.5 Semantic Annotations for WSDL

SAWSDL [16] initiative was started by W3C in April 2006 and is still in progress. It is the successor of WSDL-S. The goal of this initiative is to develop a framework to enable the semantic annotation of Web Services. It exploits the WSDL 2.0 extension mechanisms to achieve its goals. It recognizes the ambiguity when describing Web Services using WSDL, due to the fact that it is possible to have two Web Services with similar WSDL descriptions offering totally different services. They argue that this ambiguity can be resolved by adding semantic annotations inside the WSDL documents where necessary. SAWSDL is based on WSDL-S and it, likewise, offers mechanisms for referencing concepts defined in external semantic models. As a result, like WSDL-S, SAWSDL is agnostic to semantic representation languages. In addition, it enables semantic annotations for Web Services not only for discovering them but also for invoking them. SAWSDL introduces three extensibility attributes to WSDL 2.0 elements: the attribute “modelReference” that specifies the association between a WSDL component and a concept in some semantic model; this attribute is used to annotate XML Schema complex type definitions, simple type definitions, element declarations, and attribute declarations as well as WSDL interfaces, operations, and faults; finally, two additional attributes are introduced, named “liftingSchemaMapping” and “loweringSchemaMapping”, that are added to XML Schema element declarations, complex type definitions and simple type definitions for specifying mappings between semantic data and XML, that can be used during service invocation.

2.4.6 Other approaches

In [11], the author introduces a lightweight WSDL extension for the description of QoS characteristics of a Web Service. His approach is the following: he creates a WSDL metamodel in accordance to the WSDL specification (the XML schema of the specification); he introduces terms related to QoS characteristics for Web Services, using various resources from the literature; he creates classes corresponding to these terms; finally he adds these classes in his metamodel, extending the WSDL metamodel he created initially. The new metamodel is called Q-WSDL (QoS-enabled WSDL).

In [55], the authors introduce an automated software tool that uses model-driven architecture (MDA) techniques to generate an OWL-S description of a Web Service from a UML model. The transformations are performed using XSLT[10].

[10] http://www.w3.org/TR/xslt

2.5 Semantic Web Service Selection

Initiatives such as OWL-S, WSMF, WSMO, WSDL-S and SAWSDL, that were discussed in the previous paragraphs, provide a framework for semantically annotating Web Services, thus enabling the semantic discovery and selection of Web Services, omitting, however, to propose possible concrete selection mechanisms. Service selection is distinct from service discovery. Service discovery involves the discovery of Web Services published in registries such as UDDI. Service selection involves the algorithms that are followed in order to choose (select) a Web Service among a number of available Web Services after these Web Services have been discovered according to some criteria. There are numerous proposals regarding selection mechanisms in the literature and we will briefly discuss the most important ones.

The "Matchmaker" system, a semantic Web Services discovery system, is introduced in [54]. The system offers a semantic matching algorithm comparing the IOPEs (Input, Output, Precondition and Effect in Profile Ontology of OWL-S) of Web Service descriptions, stored in a repository, with those in a client's request. However, the preconditions and effects are still not efficiently integrated into the algorithm. It introduces semantic matching degrees, namely "exact" (if the requested and the advertised concepts are the same or if the requested concept is an immediate subclass of the advertised concept), "plugIn" (if the advertised concept subsumes the requested concept, then the advertised concept is assumed to encompass the requested one or in other words the advertised concept can be plugged instead of the requested one), "subsumes" (if the requested concept subsumes the advertised one, then the provider may or may not completely satisfy the requester) and "fail" (no subsumption relation between the requested and the advertised concepts). The search algorithm is based on a capability-based search mechanism [53], enabled by

OWL-S. However, the algorithm is primarily focused on the semantic similarities between Web Service descriptions and requests containing a single input and a single output. As a result, it lacks in handling multiple inputs and outputs and it is not able to provide alternative Web Services that may match (partially or totally) the request.

The algorithm introduced in [54] is extended and enhanced by its creators in [52]. They present an OWL-S Integrated Development Environment (OWL-S IDE) [51], an eclipse-based development environment, which provides a development and execution environment for OWL-S. The tool combines existing Web Service frameworks with semantic web frameworks. It supports development of OWL-S descriptions, as well as advertisement, discovery, and execution of OWL-S Web Services. They extend UDDI registry with OWL-S discovery features. The enhanced algorithm includes the newly introduced OWL-S service product and service classification properties. The service classification property is used to represent the categories to which Web Services belong, utilizing OWL concepts to represent their categories as opposed to syntactic codes (string-based) used in UDDI, thus offering more efficient matching.

In [42], an annexed algorithm is proposed that extends the algorithm presented in [54], by arranging the returned Web Service descriptions according to the usability of these Web Services. It can handle multiple inputs and outputs and is able to return alternative Web Service descriptions when more than one Web Services are matched against the user's request. The algorithm includes three steps: the first step involves the semantic matching of the Web Service descriptions with the client's request, and it is the same as the one used in the "Matchmaker" system; in the second step, the algorithm predicates the input usability of each Web Service by comparing the number of the required (requested) inputs against the number of the Web Service inputs, meaning that if these numbers differ most probably the Web Service will differ functionality-wise compared to the user's requested functionality. The same idea is followed in calculating the output usability of each Web Service; finally, in the third step the matched Web Service descriptions are arranged based on their semantic matching and their usability.

A conceptual distributed multi-registry service discovery architecture that supports discovery of semantic Web Service descriptions in dynamic environments is proposed in [20]. It aims to enable the deployment of a coherent, bandwidth-efficient, and robust service discovery infrastructure for both Local Area Networks (LANs) and Wide Area Networks (WANs). This work, however, is preliminary. By dynamic environments they mean surroundings that change frequently, in which both the service descriptions and service topologies may change. They acknowledge the need of utilizing semantic service descriptions when selecting services, as a more efficient and robust approach. They additionally argue that there is no coherent infrastructure for Web Service discovery that supports the contemporary needs. The proposed system basically consists of three different roles, namely client nodes, service nodes, and registry nodes, matching the three roles known from the service-oriented architecture, namely consumer, provider, and registry. These nodes of course may be inter-connected to each other. It may be also possible for nodes to engage in several roles simultaneously. They aim to build a generic, layered architecture that can be used with different registry information models and languages. Consequently, from the service selection perspective, they propose that it should be possible to use different query evaluation or matchmaking strategies, as well as registry cooperation strategies, without, however, proposing a concrete strategy.

The development of a mixed semantic Web Service discovery and composition framework is presented in [45]. The framework has been validated in the context of SAP's Guided

Procedures[11]. The framework does not attempt to fully automate all decisions. It assumes the lack of rich and accurate annotations of Web Services and client requests, thus offering assistance by suggesting solutions, identifying inconsistencies, and completing some of the user's decisions. The functions offered by the framework have the form of services. These services are namely Semantic Discovery, Semantic Dataflow Consolidation and Semantic Control Flow Consolidation. Semantics are expressed using ontologies, which are specified in OWL, while the services are described using a slightly modified fragment of OWL-S. The users can enter desirable descriptions of services using a wizard that allows users to specify desired service profile attributes (*e.g.*, IOPEs) in relation to loaded ontologies. The underlying reasoning framework is a combination of semantic reasoning functionality and of service composition planning functionality based on the GraphPlan algorithm[12].

A framework for Semantic Web Service discovery and planning is proposed in [8]. It is based on currently emerging technologies such as ontological knowledge bases, OWL, OWL-S, WSDL, Description Logic (DL)[13], *etc.* Two knowledge bases are created: a background knowledge base (a domain ontology specified in OWL) and an OWL-S Web Service knowledge base. The background knowledge base defines concepts and terms used for describing Web Services. The Web Service knowledge base stores Web Service descriptions. An agent uses the OWL-S API [49] to extract Web Service metadata, and applies a DL inference engine, called Racer [26], for reasoning with the metadata with respect to a given background knowledge base. Reasoning tasks performed by Racer include profile matchmaking, input/output subsumption testing (comparing and matching them semantically), and preconditions/effects analysis, which are basic mechanisms for Web Services discovery

[11]http://www.sap.com/solutions/netweaver/cafindex.epx
[12]http://www.cs.cmu.edu/ avrim/graphplan.html
[13]http://en.wikipedia.org/wiki/Description_logic

and invocation planning. The authors provide a prototype system as well.

An ontology-based rating model for service quality, facilitating the semantic Web Services discovery, is proposed in [50]. The ratings will be provided by reliable third-party organizations. Service providers will describe their services using OWL-S, which includes a mechanism for adding ratings. These descriptions will be used in matchmaking along with the service consumers' requests and preferences. Service consumers will specify their desired set of rating classifications by using rating classification terms defined in the rating model. The matching algorithm can be described as follows: a single rating classification P could be a match to a single rating classification Q in terms of exact match, specialized match, generalized match, and failed match, if P is equivalent concept to Q, P is subsumed by Q, Q is subsumed by P, and P has none of the former relations mentioned with Q exist, respectively. In addition, the service consumers can assign a priority to each rating classification in accordance with their preferences. The algorithm will return a ranked set of available services that will pass the matchmaking procedure.

Important work on service selection is also presented in [15] and [37]. A general observation from searching in the literature for semantic Web Service selection algorithms is that most research efforts are focused in specifying a framework realizing the semantic Web Service discovery and composition without paying attention to the actual selection algorithms that will be used during the Web Service selection process. Most importantly there is no de-facto standard integrating the existing technologies together for automation or semi-automation of Web Services discovery. The novelty in the current thesis, compared to the proposals in the literature, comes from adopting the MDD initiative in our framework.

Chapter 3

PIM for Carrier Applications

Model transformations are destined to be applied between source and target models. In this respect, each of these models has to conform to a schema or a domain model (metamodel). In this chapter we present the domain model that the service description source models have to conform to. The metamodel is a Platform Independent Model (PIM). A PIM is a model of a software or business system that is independent of the specific technological platform used to implement it[1].

3.1 Platform Independent Model UML Diagrams

Our PIM was designed and implemented especially for the needs of service-oriented carrier applications. The metamodel was compiled by analyzing the abstract classes of the CORBA [21], .Net [39], J2EE [5], and WSDP [40] frameworks, the EDOC [22], and IBM's Service UML profile [29]. In addition, we consulted a survey of adaptive middleware

[1]http://en.wikipedia.org/wiki/Platform-independent_model

provided in [47]. However, due to the fact that the PIM was created by analyzing and abstracting the concepts of a wide variety of technologies, some of the concepts defined in it were not needed in our framework, since we targeted, on the platform-specific side, the Web Services for carrier applications. This is the reason why we distinguish between two PIM versions namely PIM Full Version and PIM Lite Version. The lite version of our PIM is of course a subset of the full version, however, for completeness we provide the diagrams depicting both versions. Nevertheless, the full version of our PIM can be used in other research approaches dealing with carrier applications. The full version of the PIM is shown in Appendix A in figures A.1, A.2, A.3 and A.4. The PIM concepts not used in our framework will not be described in this chapter, however they are provided in Appendix A. The lite version of our PIM is shown in figure 3.1.

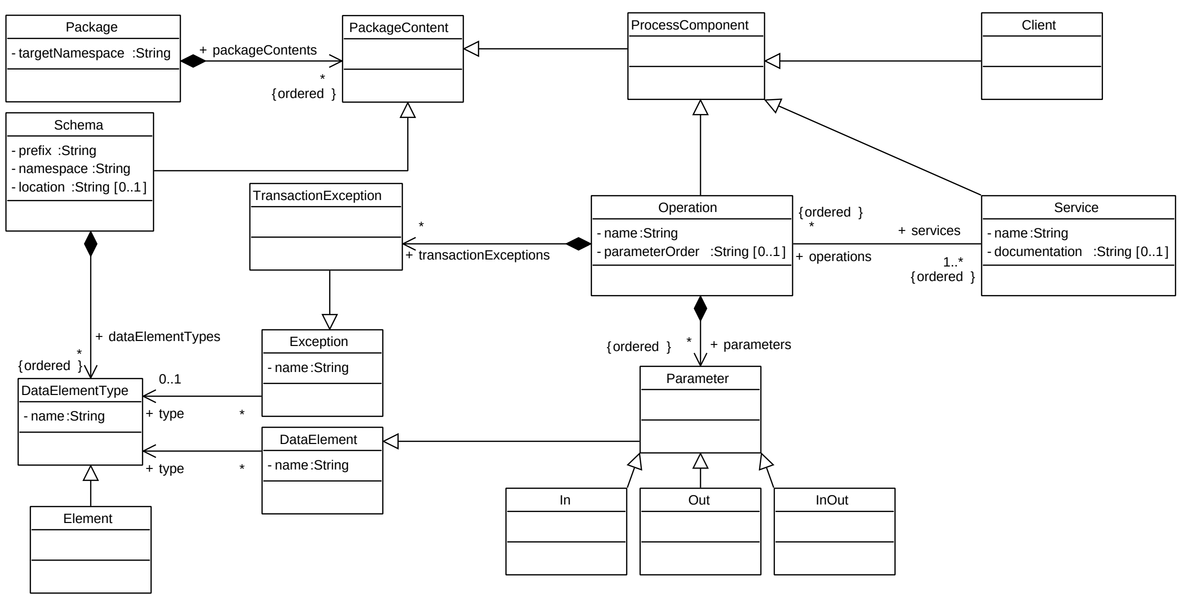


Figure 3.1: Syntactic PIM Lite Version

The root element of the PIM is the "Package" class, representing the notion of a package containing entities. The package must have a namespace attached, which is a unique identifier of its definition. A package may contain operations, services, clients (all subclasses of "ProcessComponent" class, that represents active processing entities), or schemas. A client is an entity that may query for, select, or use services. A service must have a name, and may provide a brief documentation of its role. A service contains operations, however more that one services may use an operation. An operation must have a name, and contains input, output and/or input/output parameters, and may throw transaction exceptions. An operation may specify the order of its parameters, when it is used with an RPC-binding, because it might be useful to be able to capture the original RPC function signature. Both parameters and transaction exceptions must have a name and a type. We made the exception type attribute optional in case someone will introduce a different type of exception besides transaction exceptions in the future. However, for a transaction exception, the type attribute is necessary. The parameter and transaction exception types are contained in schemas. A schema must have a prefix, a namespace and may indicate the physical location of the file containing it. If the location is omitted, it means the schema is available on the World Wide Web. This model can be extended for a specific application domain by subclassing the "Service" class. For example, for carrier applications we can specialize the "Service" class with services obtained from the Parlay X specification of Web services for the telecom domain.

We have to make clear that the domain model (metamodel), shown in figure 3.1, as well as its full version presented in Appendix A, describes the syntactic (functional) entities of our framework. The metamodel specifying the semantic (non-functional) entities is presented in Chapter 4. The two metamodels (syntactic and semantic) are complementary and should be considered as one.

3.2 Platform Independent Model Documentation

We will now provide the documentation of the lite version of our PIM to facilitate the comprehension of the proposed model.

Table 3.1: Syntactic PIM Lite Version Documentation

Class Name	**Package**.
Semantics	Defines a structural container for "top level" model elements.
Extends	None.
Attributes	***targetNamespace: String (required)*** The namespace of the service definition.
Associations	***PackageContent (zero or more)*** The model element(s) within the package.
Class Name	**PackageContent (abstract)**.
Semantics	An abstract capability that represents an element that may be placed in a package and thus referenced from other elements of the package.
Extends	None.
Attributes	None.
Associations	None.
Class Name	**ProcessComponent (abstract)**.
Semantics	A ProcessComponent represents an abstract active processing unit (it does something).
Extends	***PackageContent***.
Attributes	None.
Associations	None.
Class Name	**Exception (abstract)**.
Semantics	When defining a service it is useful to declare the exceptions that may be thrown or events that may occur as a result of an erroneous state.

Continued on next page

Table 3.1 – continued from previous page

Extends	None.
Attributes	***name: String (required)*** The name of the exception.
Associations	***DataElementType (zero or one)*** When the exception is a data type then this association is used to define its type (*e.g.*, XML schema element, simpleType, complexType *etc.*).
Class Name	**TransactionException**.
Semantics	The exceptions thrown during the execution of transactions.
Extends	***Exception***.
Attributes	None.
Associations	None.
Class Name	**Schema**.
Semantics	Represents an external imported XML schema declaration to be used inside a WSDL document. The elements of a schema are used when defining a service, for example when defining the parameters of the operations.
Extends	***PackageContent***.
Attributes	***prefix: String (required)*** The prefix used when referencing the schema. ***namespace: String (required)*** The URI representing the "targetNamespace" attribute of a schema. ***location: String (optional)*** When importing a schema as an external document the "location" attribute is used to indicate the location, in the local file system, of the file containing the schema. If it is omitted it means that the schema is accessible over the Internet.
Associations	***DataElementType (zero or more)*** The XML schema "element" element(s) declared inside the schema. These elements will be used when defining the service.

Continued on next page

Table 3.1 – continued from previous page

Class Name	**Element**.
Semantics	Represents the "element" element in an XML schema document. The "complexType" and "simpleType" elements define data types, not actual data elements. The distinction between these two is analogous to the difference between a class and an instance of that class. The data elements are defined using the "element" element.
Extends	***DataElementType***.
Attributes	None.
Associations	None.
Class Name	**Service**.
Semantics	In a service-oriented architecture, we need a clear understanding of the term service. This is achieved by defining the class Service.
Extends	***ProcessComponent***.
Attributes	***name: String (required)*** The name of the service. ***documentation: String (optional)*** A brief documentation (description) of the service.
Associations	***Operation (zero or more)*** The operation(s) of the service. ***PreConditions (zero or one)*** The preconditions of the service that need to be satisfied before the service is executed. This is part of the semantic information of the service and as a result the association is shown in the semantic metamodel presented in chapter 4, in figure 4.1. ***ServiceContext (zero or one)*** The context of the service. This is part of the semantic information of the service and as a result the association is shown in the semantic metamodel presented in chapter 4, in figure 4.1. ***PointsOfAvailability (zero or more)*** The location(s) the service is available. This is part of the semantic information of the service and as a result the association is shown in the semantic metamodel presented in chapter 4, in figure 4.2.

Continued on next page

Table 3.1 – continued from previous page

	ServiceLocation (zero or one) The physical location of the service. This is part of the semantic information of the service and as a result the association is shown in the semantic metamodel presented in chapter 4, in figure 4.2.
Class Name	**Operation**.
Semantics	Represents an operation of a service.
Extends	***ProcessComponent***.
Attributes	***name: String (required)*** The name of the operation. ***parameterOrder: String (optional)*** Operations do not specify whether they are to be used with RPC-like bindings or not. However, when using an operation with an RPC-binding, it is useful to be able to capture the original RPC function signature. For this reason, an operation may specify an order of parameter names via the "parameterOrder" attribute. The value of the attribute is a list of message part names separated by a single space. Note that this information serves as a "hint" and may safely be ignored by those not concerned with RPC signatures. Also, it is not required to be present, even if the operation is to be used with an RPC-like binding.
Associations	***Service (one or more)*** The service(s) containing the operation. ***Parameter (zero or more)*** The parameter(s) associated with the operation. ***TransactionException (zero or more)*** The transaction exception(s) associated with the operation.
Class Name	**DataElement (abstract)**.
Semantics	DataElement is the abstract super type of all parameters and global data sources defined and used by the service. It defines some kind of information.
Extends	None.
Attributes	***name: String (required)*** The name of the data element.
	Continued on next page

Table 3.1 – continued from previous page

Associations	***DataElementType (exactly one)*** The type of the data element.
Class Name	**DataElementType (abstract)**.
Semantics	DataElementType is the abstract super type of all elements that can be types of a DataElement.
Extends	None.
Attributes	***name: String (required)*** The name of the type.
Associations	None.
Class Name	**Parameter (abstract)**.
Semantics	The abstract super type of the parameters of an operation.
Extends	***DataElement***.
Attributes	None.
Associations	None.
Class Name	**InOut**.
Semantics	Represents the "in/out" parameters. If a parameter appears in both the input and output, it is an "in/out" parameter. The value of an "in/out" argument is sent in the input and is modified from the reply. An "in/out" argument is therefore both an "in" and "out" argument.
Extends	***Parameter***.
Attributes	None.
Associations	None.
Class Name	**In**.
Semantics	Represents the "in" parameters. If a parameter appears in only the input, it is an "in" parameter. "In" arguments are sent in the input but do not change as a result of the method invocation. This is a direct mapping of the pass-by-value semantics of arguments in Java method calls.
Extends	***Parameter***.
Attributes	None.
Associations	None.

Continued on next page

Table 3.1 – continued from previous page

Class Name	**Out**.
Semantics	Represents the "out" parameters. If a parameter appears in only the output message, it is an "out" parameter. "Out" arguments appear in the method signature, but their value is not sent with the input message. However, a new value for the argument may appear in the response and, if so, the argument is modified from the returned value.
Extends	***Parameter***.
Attributes	None.
Associations	None.
Class Name	**Client**.
Semantics	Represents the possible clients that may query for or use a service.
Extends	***ProcessComponent***.
Attributes	None.
Associations	***ClientProfile (zero or one)*** The profile of the client. This is part of the semantic information of the client and as a result the association is shown in the semantic metamodel presented in chapter 4, in figure 4.1.

Chapter 4

Semantic Service Description Framework

The framework proposed in this dissertation allows for the specification of both syntactic and semantic descriptions of services. The syntactic side was presented in Chapter 3. We will now present the semantic service description framework.

4.1 Context Definition

The framework is expressed with a domain model (metamodel) containing all the semantic information. The ultimate goal of the metamodel is to add context-awareness in our framework in both the client and the service side. However, context is a term quite abstract and its interpretation varies according to the application domain. Several definitions for context exist in literature. A very concise definition is given in [13], in which context is defined as "any information that can be used to characterize the situation of entities (*i.e.*, whether a person, place or object) that is considered relevant to the interaction between a user and an application, including the user and the application themselves".

4.2 Semantic Description UML Diagrams

In accordance to the context definition presented in the previous paragraph and by consulting the OMG's "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms" [24], the W3C's "QoS for Web Services: Requirements and Possible Approaches" [33], IBM's article "Understanding quality of service for Web services" [35] and W3C's article "Enabling Open, Interoperable, and Smart Web Services: The Need for Shared Context" [34], we compiled the metamodel presented in this chapter. Keep in mind that this metamodel is complementary to the one presented in Chapter 3, and that these two metamodels should be considered as one. In addition, although we provide a quite long list of semantic-oriented properties, we must clarify that our purpose is not to provide an exhaustive list of semantic-oriented properties but to provide the framework for anyone who wishes to update/modify this list according to his/her needs. This is why we propose an easily extensible metamodel.

The proposed metamodel depicting the properties of our semantic service selection framework is shown in figures 4.1, 4.2, 4.3 and 4.4.

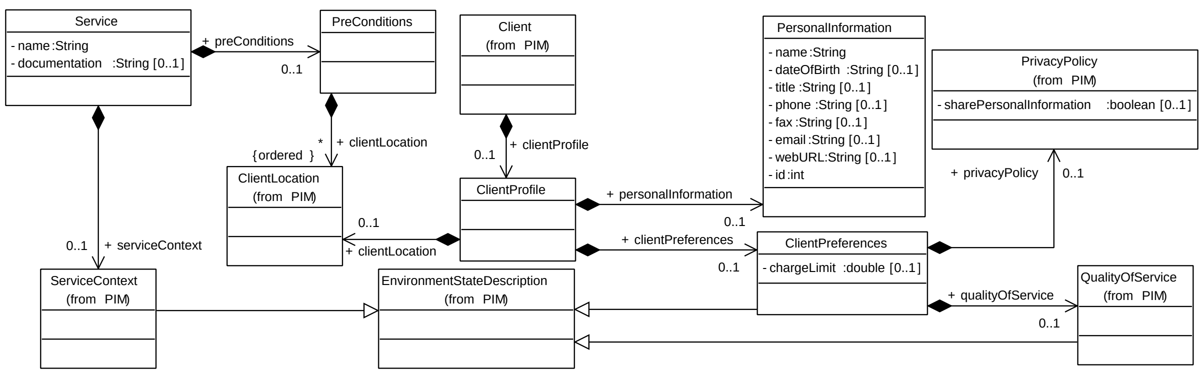


Figure 4.1: Semantic PIM Part 1

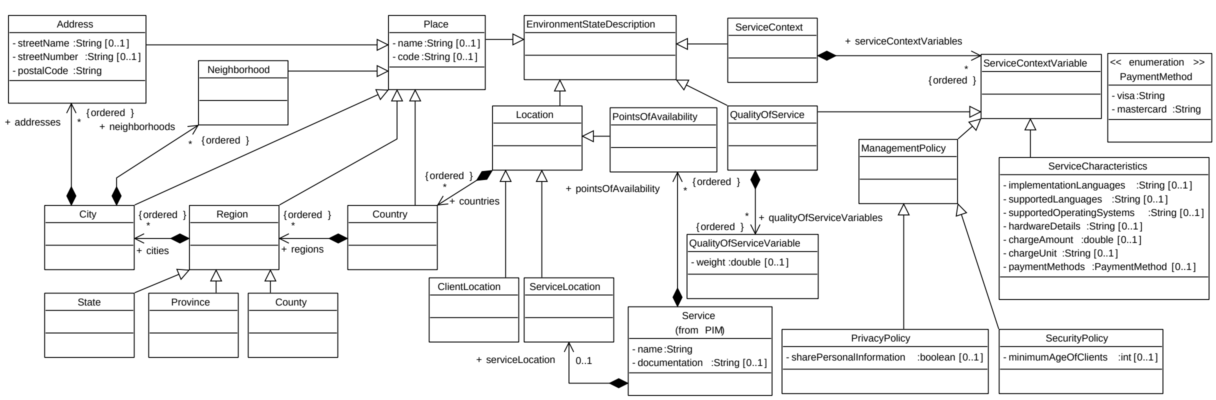


Figure 4.2: Semantic PIM Part 2

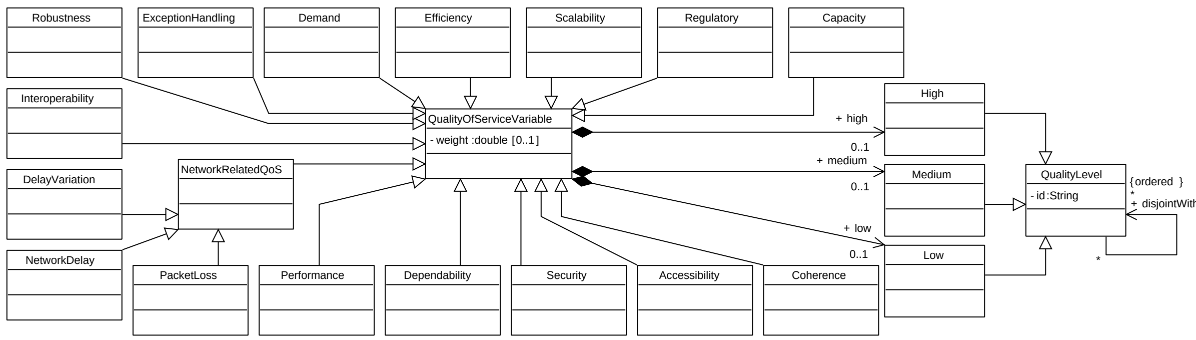


Figure 4.3: Semantic PIM Part 3

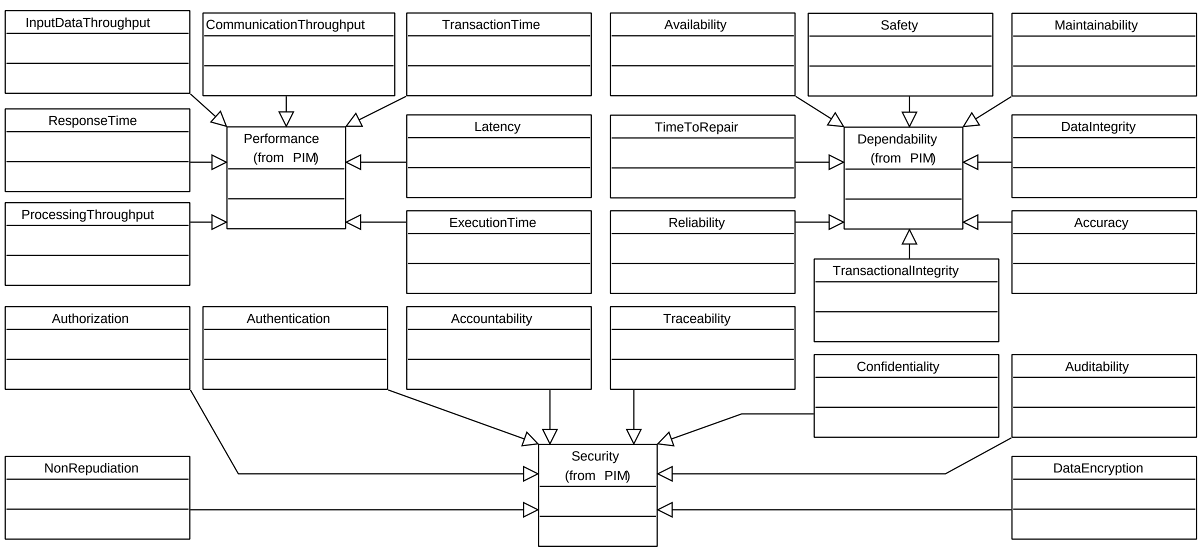


Figure 4.4: Semantic PIM Part 4

4.3 Semantic PIM Documentation

We will now provide the documentation of the metamodel, shown in the previous paragraph, to facilitate the comprehension of the proposed model.

A client may specify a profile containing his/her location, personal information (*e.g.*, name, phone, *etc.*), the maximum amount a service should charge, and/or personal preferences. A client's personal preferences may include his/her desired quality of service characteristics and/or his/her desired privacy policies. The specification of the desired QoS characteristics is achieved by choosing the appropriate QoS characteristics, selecting the quality level of these characteristics (one of high, medium or low) and assigning a weight to each characteristic, representing the degree of importance this characteristic has for the client. A client's preferable privacy policies include policies a service should satisfy, such as sharing its clients' personal information with other services, to avoid using services violating his/her desired privacy policies.

A service provider may choose to specify preconditions, when describing a service, or attach some context to the service's description. For now, the preconditions include only the allowed client locations, however, it is very easy to extend the metamodel to accommodate new preconditions as needed. The service's context includes a number of properties (variables). A service provider may specify the service's characteristics, such as the service's charge amount, the payment methods *etc.*, or attach management policies for the service. We have defined two types of management policies namely privacy policies and security policies. A service description may of course include the QoS characteristics (variables) of the service. Each characteristic is associated with a quality level (high, medium or low), or a combination of quality levels (low and medium, medium and high *etc.*). Furthermore, a service description may include the physical location of the service and/or the points of availability of the service. The attributes of the "location" property, used by both the service providers and the clients, were compiled from the Where Am I Language (WAIL) [38]. The level of granularity WAIL provides is at the level of a street address. This is only a proposed way to specify the location. For example, we might have used coordinates to specify the location. Keep in mind that it is very easy to modify our framework to include any desired changes. We have provided a quite long list of QoS characteristics, compiled from the literature. However, it is not our intention to provide an exhaustive list of QoS variables, but to provide the framework for specifying QoS variables.

The framework offers extension points facilitating the specification of new QoS properties or policies as needed. As shown in figure 4.3, each QoS property is a subclass of the abstract class "QualityOfServiceVariable". In this respect, any new QoS property can be introduced in our framework by representing the desired QoS property as a class and by specifying it as a subclass of "QualityOfServiceVariable" class. The selection policies are hardcoded in our framework, however it is very easy to introduce new policies since we used the "Factory" design pattern to specify our existing policies. New policies can be introduced by specifying a class containing the new policy algorithm (implementing our "Policy" interface class). After specifying the policy class we only need to specify a class calling the desired policy (implementing our "PolicyCreator" interface class). With this modular approach the introduction of new policies is facilitated to a great extent. Another approach to introduce new policies could have been the subclassing of the abstract class "ManagementPolicy" shown in figure 4.2.

Notice that by using our framework, both the service providers and the clients, use a common terminology for their descriptions. This enables the framework to query over existing service descriptions and choose the most appropriate services according to the profile of a client. The formal description of the metamodel is shown in table 4.1.

Table 4.1: Semantic PIM Documentation

Class Name	**ClientProfile**.
Semantics	The profile of a client.
Extends	None.
Attributes	None.
Associations	***ClientLocation (zero or one)*** The physical location of the client. ***PersonalInformation (zero or one)*** The personal information of the client. ***ClientPreferences (zero or one)*** The preferences of the client.
Class Name	**PersonalInformation**.
Semantics	The personal information of a client.
Extends	None.
Attributes	***name: String (required)*** The name of a client. ***dateOfBirth: String (optional)*** The date of birth of a client. ***title: String (optional)*** The title (*e.g.*, profession, client category, role) of a client. ***phone: String (optional)*** The phone number of a client. ***fax: String (optional)*** The fax number of a client. ***email: String (optional)*** The email address of a client. ***webURL: String (optional)*** The personal web page of a client. ***id: Integer (required)*** The identification number of a client, used for verification purposes.

Continued on next page

Table 4.1 – continued from previous page

Associations	None.
Class Name	**EnvironmentStateDescription (abstract)**.
Semantics	It is an abstraction of all elements that form (describe) the state of the environment, such as the context of a service, the preferences of a client *etc*.
Extends	None.
Attributes	None.
Associations	None.
Class Name	**ClientPreferences**.
Semantics	The preferences of a client.
Extends	***EnvironmentStateDescription***.
Attributes	***chargeLimit: double (optional)*** The maximum amount a service should charge.
Associations	***PrivacyPolicy (zero or one)*** The desired privacy policies that a service should satisfy. ***QualityOfService (zero or one)*** The desired QoS properties a service should satisfy. This would be the ideal configuration of a service.
Class Name	**Preconditions**.
Semantics	The preconditions that need to be satisfied before the execution of the service.
Extends	None.
Attributes	None.
Associations	***ClientLocation (zero or more)*** Clients having these physical locations are allowed to have access to the service.
Class Name	**ServiceContext**.
Semantics	The information related to the context of the service.
Extends	***EnvironmentStateDescription***.
Attributes	None.
Associations	***ServiceContextVariable (zero or more)*** The variable(s) composing the context of a service.

Continued on next page

Table 4.1 – continued from previous page

Class Name	**ServiceContextVariable (abstract)**.
Semantics	An abstraction of the variables belonging to the context of a service.
Extends	None.
Attributes	None.
Associations	None.
Class Name	**ServiceCharacteristics**.
Semantics	A list of simple characteristics of a service.
Extends	***ServiceContextVariable***.
Attributes	***implementationLanguages: String (optional)*** The programming languages used to implement the service (*e.g.*, Java). ***supportedLanguages: String (optional)*** The languages that the service interface supports (*e.g.*, English). ***supportedOperatingSystems: String (optional)*** The operating systems the service supports (*e.g.*, Windows XP). ***hardwareDetails: String (optional)*** Details about the hardware the service's software runs on top. ***chargeAmount: double (optional)*** The amount the service charges per use. ***chargeUnit: String (optional)*** The money unit of the amount specified in the previous attribute (*e.g.*, euro). ***paymentMethods: PaymentMethod (optional)*** The possible payment methods (visa or mastercard in our model).
Associations	None.
Class Name	**ManagementPolicy (abstract)**.
Semantics	A management policy is a set of rules that is specified by a user or a computing entity to restrict or guide the execution of actions. For example, in the context of system security, a system administrator may use policies to define who has the right to execute what services; in the context of privacy protection, a user may use policies to restrict the type of personal information that can be shared by the public services.
Extends	***ServiceContextVariable***.
Attributes	None.
Associations	None.

Continued on next page

Table 4.1 – continued from previous page

Class Name	**PrivacyPolicy**.
Semantics	Policies regarding the handling of personal information of the clients.
Extends	***ManagementPolicy***.
Attributes	***sharePersonalInformation: Boolean (optional)*** Indicates whether the service should be allowed to share the clients' personal information.
Associations	None.

Class Name	**SecurityPolicy**.
Semantics	Policies specifying groups of users who are not allowed to use the service.
Extends	***ManagementPolicy***.
Attributes	***minimumAgeOfClients: Integer (optional)*** Indicates the minimum age of the clients that are allowed to use the service.
Associations	None.

Class Name	**Location (abstract)**.
Semantics	An abstraction of all the elements denoting location.
Extends	***EnvironmentStateDescription***.
Attributes	None.
Associations	***Country (zero or more)*** A list of countries forming the location of an entity. The "country" element is the root of a series of elements (*e.g.*, province, city, address *etc.*) describing the location of an entity.

Class Name	**ClientLocation**.
Semantics	The location of a client.
Extends	***Location***.
Attributes	None.
Associations	None.

Class Name	**ServiceLocation**.
Semantics	The location of a service.
Extends	***Location***.
Attributes	None.
Associations	None.

Continued on next page

Table 4.1 – continued from previous page

Class Name	**PointsOfAvailability**.
Semantics	The locations the service is available.
Extends	***Location***.
Attributes	None.
Associations	None.

Class Name	**Place (abstract)**.
Semantics	An abstraction denoting anything which can be considered a place, such as a city.
Extends	***EnvironmentStateDescription***.
Attributes	***name: String (optional)*** A name by which this place is known. ***code: String (optional)*** A well-known code by which this place is known; when applied to a Country, the value must be an ISO 3166-1 country code; when applied to a Region, the value must be an ISO 3166-2 region code.
Associations	None.

Class Name	**Country**.
Semantics	An internationally-recognized country; anything which has an ISO country code.
Extends	***Place***.
Attributes	None.
Associations	***Region (zero or more)*** A list of regions contained in the country.

Class Name	**Region (abstract)**.
Semantics	A subdivision of a Country with a well-known name and area.
Extends	***Place***.
Attributes	None.
Associations	***City (zero or more)*** A list of cities contained in the region.

Class Name	**State**.
Semantics	A type of Region used in federal systems such as the United States.
Extends	***Region***.
Attributes	None.

Continued on next page

Table 4.1 – continued from previous page

Associations	None.
Class Name	**Province**.
Semantics	A type of Region used in federal systems such as Canada.
Extends	***Region***.
Attributes	None.
Associations	None.
Class Name	**County**.
Semantics	A type of Region; usually a subdivision of a larger Region containing several Cities.
Extends	***Region***.
Attributes	None.
Associations	None.
Class Name	**City**.
Semantics	A subdivision of Region corresponding to a center of population, such as a city, town, village, etc. Does not necessarily correspond to an actual municipal government.
Extends	***Place***.
Attributes	None.
Associations	***Neighborhood (zero or more)*** A list of neighborhoods contained in the city. ***Address (zero or more)*** A list of addresses contained in the city.
Class Name	**Neighborhood**.
Semantics	A subdivision of a City.
Extends	***Place***.
Attributes	None.
Associations	None.
Class Name	**Address**.
Semantics	A specific home, office, apartment, place of business, *etc.*
Extends	***Place***.
Attributes	***streetName: String (optional)*** The name of the street.

Continued on next page

Table 4.1 – continued from previous page

	streetNumber: String (optional) The number of the street. ***postalCode: String (required)*** The postal code of the address.
Associations	None.

Class Name	**QualityOfService**.
Semantics	A container of all the QoS variables.
Extends	***EnvironmentStateDescription***, ***ServiceContextVariable***.
Attributes	None.
Associations	***QualityOfServiceVariable (zero or more)*** The contained list of QoS variables.

Class Name	**QualityOfServiceVariable (abstract)**.
Semantics	An abstraction of all QoS variables. We have to clarify that all QoS variables, in our framework, are measured upon the quality level they are offered (one or more values from "High", "Medium", or "Low"), regardless the typical unit each QoS variable may be measured upon. When the standard measurement unit for a QoS variable is provided in its description, it is provided for the better understanding of the QoS variable itself.
Extends	None.
Attributes	***weight: double (optional)*** The weight a client can attach to a QoS variable denoting the importance of the specific variable for the client. The attribute is used in the service selection process performed by the framework.
Associations	***High (zero or one)*** Denotes high quality level. ***Medium (zero or one)*** Denotes medium quality level. ***Low (zero or one)*** Denotes low quality level.

Continued on next page

Table 4.1 – continued from previous page

Class Name	**Performance (abstract)**.
Semantics	The performance of a web service represents how fast a service request can be completed. It can be measured in terms of throughput (the number of web service requests served in a given time interval), response time, latency, execution time, transaction time, and so on. In general, high quality web services should provide higher throughput, faster response time, lower latency, lower execution time, and faster transaction time.
Extends	***QualityOfServiceVariable***.
Attributes	None.
Associations	None.
Class Name	**InputDataThroughput**.
Semantics	Represents the arrival rate of user data input channel, software or hardware, averaged over a time interval. The rate unit for this throughput is bit/sec.
Extends	***Performance***.
Attributes	None.
Associations	None.
Class Name	**CommunicationThroughput**.
Semantics	Represents the rate of user data output to a channel averaged over a time interval. The rate unit for this throughput is bit/sec.
Extends	***Performance***.
Attributes	None.
Associations	None.
Class Name	**ProcessingThroughput**.
Semantics	Represents the amount of processing able to be performed in a period of time. The unit of rate is instructions/sec.
Extends	***Performance***.
Attributes	None.
Associations	None.
Class Name	**ResponseTime**.
Semantics	The time required to complete a web service request.
Extends	***Performance***.

Continued on next page

Table 4.1 – continued from previous page

Attributes	None.
Associations	None.
Class Name	**Latency**.
Semantics	The round-trip delay (RTD) between sending a request and receiving the response.
Extends	***Performance***.
Attributes	None.
Associations	None.
Class Name	**TransactionTime**.
Semantics	Represents the time that passes while the web service is completing one complete transaction. This transaction time may depend on the definition of web service transaction.
Extends	***Performance***.
Attributes	None.
Associations	None.
Class Name	**ExecutionTime**.
Semantics	The time taken by a web service to process its sequence of activities.
Extends	***Performance***.
Attributes	None.
Associations	None.
Class Name	**Dependability (abstract)**.
Semantics	Dependability is the property of computer systems such that reliance can justifiably be placed on the service it delivers. It includes QoS characteristics such as: availability, reliability, safety, maintainability, accuracy and integrity.
Extends	***QualityOfServiceVariable***.
Attributes	None.
Associations	None.

Continued on next page

Table 4.1 – continued from previous page

Class Name	**Availability**.
Semantics	Availability is the quality aspect of whether the Web Service is present or ready for immediate use. It represents the probability that a service is available. Larger values represent that the service is always ready to use while smaller values indicate unpredictability of whether the service will be available at a particular time.
Extends	***Dependability***.
Attributes	None.
Associations	None.
Class Name	**TimeToRepair**.
Semantics	Time-to-repair (TTR) is associated with availability . TTR represents the time it takes to repair a service that has failed. Ideally smaller values of TTR are desirable.
Extends	***Dependability***.
Attributes	None.
Associations	None.
Class Name	**Reliability**.
Semantics	Web services should be provided with high reliability. Reliability represents the ability of a web service to perform its required functions under stated conditions for a specified time interval. The reliability is the overall measure of a web service to maintain its service quality. The overall measure of a web service is related to the number of failures per day, week, month, or year. Reliability is also related to the assured and ordered delivery for messages being transmitted and received by service requestors and service providers. (Associated with Maturity and Recoverability).
Extends	***Dependability***.
Attributes	None.
Associations	None.
Class Name	**Safety**.
Semantics	Expresses how safe the use of the Web Service is.
Extends	***Dependability***.
Attributes	None.
Associations	None.

Continued on next page

Table 4.1 – continued from previous page

Class Name	**Maintainability**.
Semantics	Expresses how well the service is maintained.
Extends	***Dependability***.
Attributes	None.
Associations	None.
Class Name	**Accuracy**.
Semantics	Web services should be provided with high accuracy. Accuracy here is defined as the error rate generated by the Web Service. The number of errors that the service generates over a time interval should be minimized.
Extends	***Dependability***.
Attributes	None.
Associations	None.
Class Name	**DataIntegrity**.
Semantics	Integrity for web services should be provided so that a system or component can prevent unauthorized access to, or modification of, computer programs or data. Data integrity defines whether the transferred data is modified in transit.
Extends	***Dependability***.
Attributes	None.
Associations	None.
Class Name	**TransactionalIntegrity**.
Semantics	Integrity for web services should be provided so that a system or component can prevent unauthorized access to, or modification of, computer programs or data. Transactional integrity refers to a procedure or set of procedures, which is guaranteed to preserve database integrity in a transaction.
Extends	***Dependability***.
Attributes	None.
Associations	None.
Class Name	**Coherence**.
Semantics	Coherence includes characteristics about concurrent and temporal consistency of data and software elements.

Continued on next page

Table 4.1 – continued from previous page

Extends	***QualityOfServiceVariable***.
Attributes	None.
Associations	None.
Class Name	**Capacity**.
Semantics	Web services should be provided with the required capacity. Capacity is the limit of the number of simultaneous requests, which should be provided with guaranteed performance. Web services should support the required number of simultaneous connections.
Extends	***QualityOfServiceVariable***.
Attributes	None.
Associations	None.
Class Name	**Scalability**.
Semantics	Sometimes the same service is not produced with the same quality level when the number of software elements increase. The capacity of software elements is limited to a minimum and maximum number of elements. Scalability refers to the ability to consistently serve the requests despite variations in the volume of requests. Web services should be provided with high scalability. Scalability represents the capability of increasing the computing capacity of service providers computer system and systems ability to process more users requests, operations or transactions in a given time interval. It is also related to performance. Web services should be scalable in terms of the number operations or transactions supported.
Extends	***QualityOfServiceVariable***.
Attributes	None.
Associations	None.
Class Name	**Efficiency**.
Semantics	The capability of the software to produce their results with the minimum resource consumption.
Extends	***QualityOfServiceVariable***.
Attributes	None.
Associations	None.

Continued on next page

Table 4.1 – continued from previous page

Class Name	**Demand**.
Semantics	Demand is the characterization of how much of a resource or a service is needed.
Extends	***QualityOfServiceVariable***.
Attributes	None.
Associations	None.
Class Name	**Robustness**.
Semantics	Web services should be provided with high robustness. Robustness represents the degree to which a web service can function correctly even in the presence of invalid, incomplete or conflicting inputs. Web services should still work even if incomplete parameters are provided to the service request invocation.
Extends	***QualityOfServiceVariable***.
Attributes	None.
Associations	None.
Class Name	**ExceptionHandling**.
Semantics	Web services should be provided with the functionality of exception handling. Since it is not possible for the service designer to specify all the possible outcomes and alternatives (especially with various special cases and unanticipated possibilities), exceptions should be handled properly. Exception handling is related to how the service handles these exceptions.
Extends	***QualityOfServiceVariable***.
Attributes	None.
Associations	None.
Class Name	**Accessibility**.
Semantics	Accessibility represents whether the web service is capable of serving the clients requests. It may be expressed as a probability measure denoting the success rate or chance of a successful service instantiation at a point in time. There could be situations when a Web service is available but not accessible. High accessibility of Web services can be achieved by building highly scalable systems.
Extends	***QualityOfServiceVariable***.
Attributes	None.

Continued on next page

Table 4.1 – continued from previous page

Associations	None.
Class Name	**Interoperability**.
Semantics	Web services should be interoperable between the different development environments used to implement services so that developers using those services do not have to think about which programming language or operating system the services are hosted on.
Extends	***QualityOfServiceVariable***.
Attributes	None.
Associations	None.
Class Name	**Security (abstract)**.
Semantics	Security is the quality aspect of the Web service of providing confidentiality and non-repudiation by authenticating the parties involved, encrypting messages, and providing access control. Web services should be provided with the required security. With the increase in the use of web services, which are delivered over the public Internet, there is a growing concern about security. The web service provider may apply different approaches and levels of providing security policy depending on the service requestor. Security for web services means providing authentication, authorization, confidentiality, accountability, traceability/auditability, data encryption, and non-repudiation.
Extends	***QualityOfServiceVariable***.
Attributes	None.
Associations	None.
Class Name	**Authorization**.
Semantics	Users (or other services) should be authorized so that they only can access the protected services.
Extends	***Security***.
Attributes	None.
Associations	None.
Class Name	**Authentication**.
Semantics	Users (or other services) who can access service and data should be authenticated.
Extends	***Security***.

Continued on next page

Table 4.1 – continued from previous page

Attributes	None.
Associations	None.
Class Name	**Confidentiality**.
Semantics	Data should be treated properly so that only authorized users (or other services) can access or modify the data.
Extends	***Security***.
Attributes	None.
Associations	None.
Class Name	**Accountability**.
Semantics	The supplier can be hold accountable for their services.
Extends	***Security***.
Attributes	None.
Associations	None.
Class Name	**Traceability** and **Auditability**.
Semantics	It should be possible to trace the history of a service when a request was serviced.
Extends	***Security***.
Attributes	None.
Associations	None.
Class Name	**DataEncryption**.
Semantics	Data should be encrypted.
Extends	***Security***.
Attributes	None.
Associations	None.
Class Name	**NonRepudiation**.
Semantics	A user cannot deny requesting a service or data after the fact. The service provider needs to ensure these security requirements.
Extends	***Security***.
Attributes	None.
Associations	None.

Continued on next page

Table 4.1 – continued from previous page

Class Name	**NetworkRelatedQoS (abstract)**.
Semantics	To achieve desired QoS for web services, the QoS mechanisms operating at the web service application level must operate together with the QoS mechanisms operating in the transport network, which are rather independent of the application. In particular, application level QoS parameters should be mapped appropriately to corresponding network level QoS parameters. Basic network level QoS parameters include network delay, delay variation, and packet loss.
Extends	***QualityOfServiceVariable***.
Attributes	None.
Associations	None.
Class Name	**NetworkDelay**.
Semantics	The average length of time a packet traverses in a network. The network delay can be handled by a good network design that minimizes the number of hops encountered and by the advent of faster switching devices like Layer 3 switches and tag switching system such as MPLS systems and ATM switches.
Extends	***NetworkRelatedQoS***.
Attributes	None.
Associations	None.
Class Name	**DelayVariation**.
Semantics	The variation in the inter-packet arrival time (leading to gaps, known as jitter, between packets) as introduced by the variable transmission delay over the network. Removing jitter requires collecting packets in buffers and holding them long enough to allow the slowest packets to arrive in time to be played in correct sequence. Jitter buffers may cause additional delay, which is used to remove the packet delay variation as each packet transits the network.
Extends	***NetworkRelatedQoS***.
Attributes	None.
Associations	None.

Continued on next page

Table 4.1 – continued from previous page

Class Name	**PacketLoss**.
Semantics	The Internet does not guarantee delivery of packets. Packets will be dropped under peak loads and during periods of congestion. Approaches used to compensate for packet loss include replay of the last packet, and transmission of redundant information. Out of order packets may need to be re-ordered at the receiver.
Extends	***NetworkRelatedQoS***.
Attributes	None.
Associations	None.
Class Name	**Regulatory**.
Semantics	Regulatory is the quality aspect of the Web service in conformance with the rules, the law, compliance with standards, and the established service level agreement. Web services use a lot of standards such as SOAP, UDDI, and WSDL. Strict adherence to correct versions of standards (for example, SOAP version 1.2) by service providers is necessary for proper invocation of Web services by service requestors.
Extends	***QualityOfServiceVariable***.
Attributes	None.
Associations	None.
Class Name	**QualityLevel (abstract)**.
Semantics	An abstraction of the available quality levels for QoS variables.
Extends	None.
Attributes	***id: String (required)*** The id for a specific quality level of a QoS variable. It must have the form "QoS_Name"+"QualityLevel" (*e.g.*, "SecurityHigh").
Associations	***QualityLevel (zero or more)*** When a service provider wishes to indicate that a specific quality level of a QoS variable is disjoint with a set of other QoS variables, then he/she uses the "disjointWith" association.
Class Name	**High**.
Semantics	Represents the high quality level.
Extends	***QualityLevel***.
Attributes	None.
Associations	None.

Continued on next page

Table 4.1 – continued from previous page

Class Name	**Medium**.
Semantics	Represents the medium quality level.
Extends	***QualityLevel***.
Attributes	None.
Associations	None.
Class Name	**Low**.
Semantics	Represents the low quality level.
Extends	***QualityLevel***.
Attributes	None.
Associations	None.

Chapter 5

PSM for Web Services in Carrier Applications

In this chapter, we present the domain model (metamodel) that the generated models of our model transformations have to conform to. In addition to model transformations, these models participate in the semantic service selection framework that will be described in Chapter 7. The metamodel is a Platform Specific Model (PSM). A platform-specific model is a model of a software or business system that is linked to a specific technological platform (*e.g.*, a specific programming language, operating system or database)[1].

5.1 Platform Specific Model UML Diagrams

Our PSM was implemented especially for the needs of Web Services in carrier applications. The current standard for describing Web Services is the Web Services Description Lan-

[1] http://en.wikipedia.org/wiki/Platform-specific_model

guage (WSDL) 1.1. WSDL 1.1 defines an XML grammar for describing network services as collections of communication endpoints capable of exchanging messages. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL provides bindings allowing to use WSDL in conjunction with other technologies such as SOAP 1.1, HTTP GET/POST, or MIME. However, WSDL is limited allowing only syntactic descriptions of Web Services. The syntactic side of our PSM was compiled by analyzing the Web Services Description Language (WSDL) 1.1 specification [9], as well as a number of book chapters related to WSDL [57], [6], [7] and a paper proposing a basic profile for WSDL [28].

WSDL is fully extensible, by providing "extensibility elements" as a mechanism to extend the language as needed. We needed, in our descriptions, the addition of semantic information into our models. We already discussed how we achieved that in the PIM discussed in chapters 3 and 4. In the PSM, we used the "extensibility elements" mechanism to extend the WSDL-specific metamodel with the semantic metamodel discussed in Chapter 4. In a nutshell, we made the root element of our semantic metamodel presented in Chapter 4 (the "EnvironmentStateDescription" class) a subclass of the class "ExtensibilityElement" of our PSM. In that way we integrated the metamodel presented in Chapter 4 into our PSM. Keep in mind of course that the classes describing the client were excluded from the PSM. Another more technical detail was that we had to make the "PreConditions" class a subclass of "ExtensibilityElement" class, because the "PreConditions" class was not a subclass of "EnvironmentStateDescription" class.

To conclude, our approach in compiling the PSM was to analyze the WSDL 1.1 specification to form the syntactic-side descriptions of the PSM and we integrated the semantic metamodel, presented in Chapter 4, in the WSDL-specific PSM using the extensibility mechanisms provided by WSDL. As a result, the semantic metamodel presented in Chapter 4 is shared by both the PIM and the PSM in our framework. The PSM is shown in figures 5.1, 5.2, 5.3, 5.4, 5.5 and 5.6.

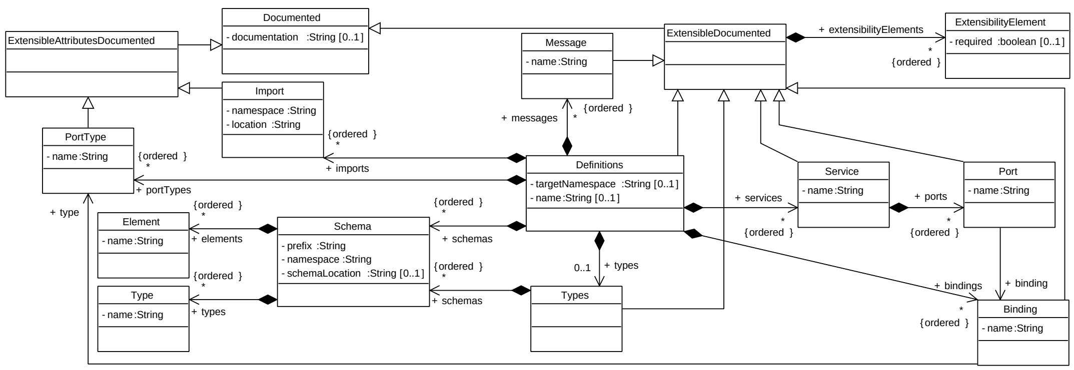


Figure 5.1: Syntactic PSM Part 1

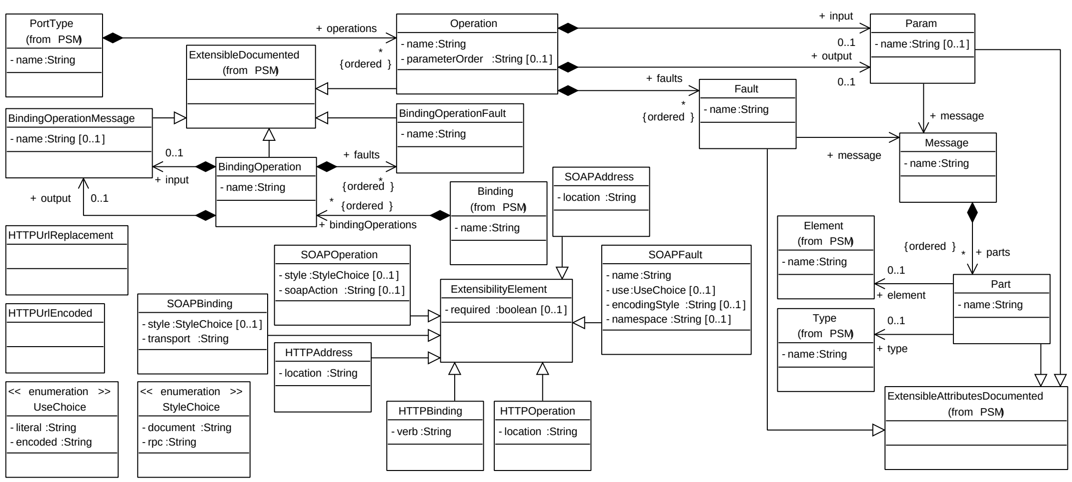


Figure 5.2: Syntactic PSM Part 2

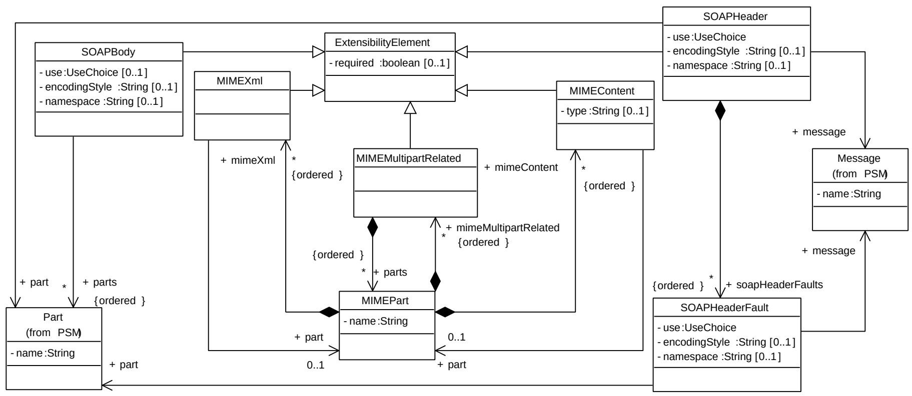


Figure 5.3: Syntactic PSM Part 3

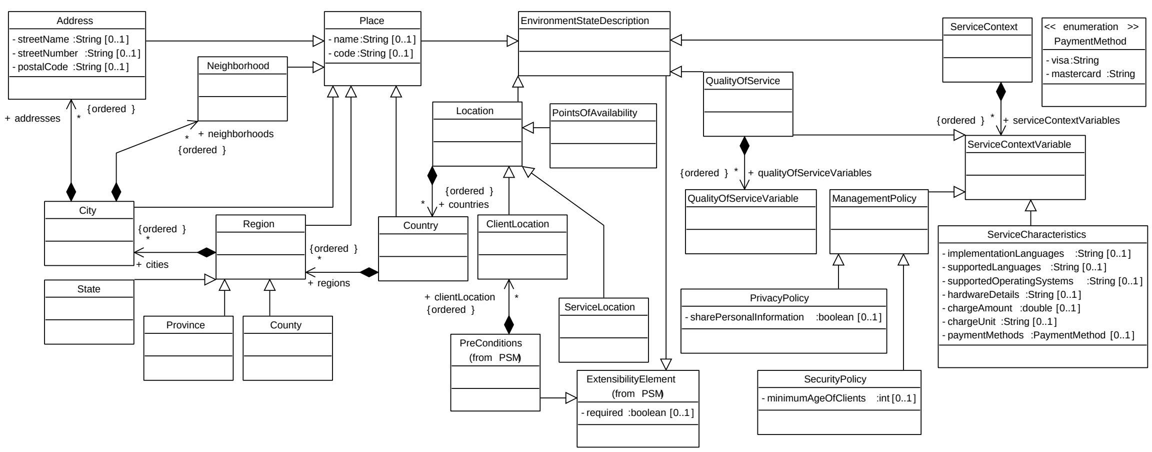


Figure 5.4: Semantic PSM Part 1

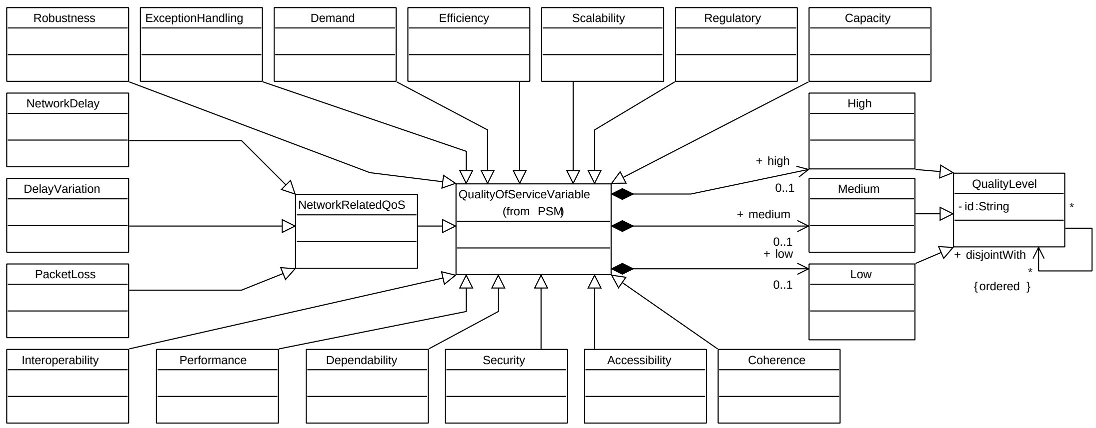


Figure 5.5: Semantic PSM Part 2

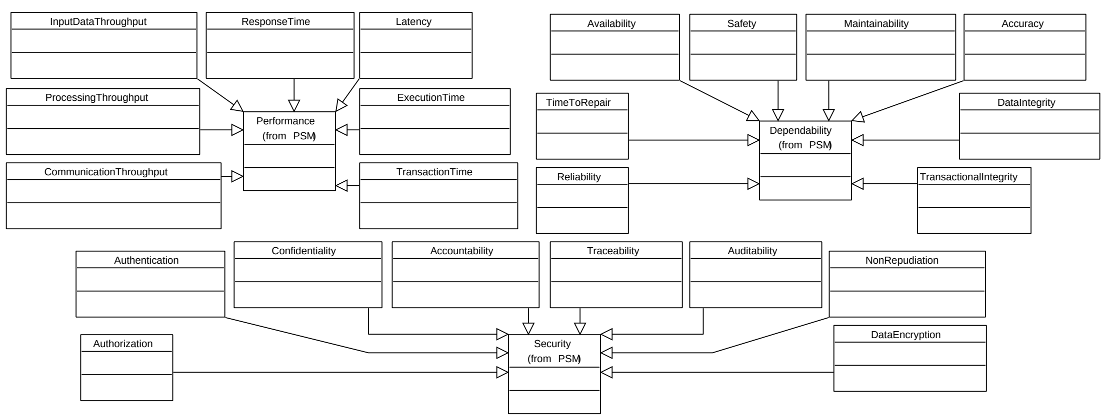


Figure 5.6: Semantic PSM Part 3

The syntactic side of our PSM essentially corresponds to the elements defined in the WSDL 1.1 specification. A WSDL document describes a Web Service in terms of the operations that it provides, the data types that each operation requires as inputs and can return in the form of results and the exceptions each operation may throw. WSDL is agnostic about the way in which the service is provided at the protocol level. In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings. Instead, the Web Service is first defined in abstract terms and then mapped onto one or more specific protocols by the use of bindings. This separation allows the reuse of abstract definitions, such as messages, which are abstract descriptions of the data being exchanged, and port types, which are abstract collections of operations. A WSDL file may also contain a set of addresses at which a bound service can be accessed. WSDL allows elements representing a specific technology (referred to as extensibility elements) under various elements defined by WSDL. The most important elements used in a WSDL document are:

- Definitions: the root element of a WSDL document.
- Types: contains customized schema definitions.
- Message: an abstraction of the exchanged data during the execution of the operations of the service.
- Operation: an abstract description of an operation supported by the service.
- Port Type: an abstract set of operations.
- Binding: a concrete protocol and data format specification for a particular port type.
- Port: a single endpoint defined as a combination of a binding and a network address.
- Service: a collection of related endpoints.
- Extensibility Element: elements representing a specific technology, extending the Web Service description.

We have defined all the semantic-specific elements that are needed inside the PSM as extensibility elements, thus enabling us to reuse and integrate the metamodel presented in Chapter 4 inside the PSM. This is shown in figure 5.4, where the root element of the semantic metamodel is defined as a subclass of the "ExtensibilityElement" class.

5.2 Platform Specific Model Documentation

We will now provide the documentation of our PSM to facilitate the comprehension of the proposed model.

Table 5.1: PSM Documentation

Class Name	**Documented (abstract)**.
Semantics	This type is extended by component types to allow them to be documented. WSDL uses this optional element as a container for human readable documentation. The content of the element is arbitrary text and elements ("mixed" in XSD). The documentation element is allowed inside any WSDL language element. In our metamodel, we handle documentation only for the "Service" element.
Extends	None.
Attributes	***documentation: String (optional)*** The documentation.
Associations	None.
Class Name	**ExtensibleAttributesDocumented (abstract)**.
Semantics	This type is extended by component types to allow attributes from other namespaces to be added.
Extends	***Documented***.
Attributes	None.
Associations	None.

Continued on next page

Table 5.1 – continued from previous page

Class Name	**ExtensibleDocumented (abstract)**.
Semantics	This type is extended by component types to allow elements from other namespaces to be added.
Extends	***Documented***.
Attributes	None.
Associations	***ExtensibilityElement (zero or more)*** The contained extensibility element(s).

Class Name	**ExtensibilityElement (abstract)**.
Semantics	This type is extended by elements from other namespaces to allow them to be added under WSDL component types.
Extends	None.
Attributes	***required: Boolean (optional)*** States whether the element is required or not.
Associations	None.

Class Name	**Definitions**.
Semantics	The root element of every WSDL file must be a <definitions> element.
Extends	***ExtensibleDocumented***.
Attributes	***name: String (optional)*** The WSDL specification describes this attribute as lightweight documentation for the content of the file. It is typically not used by software that parses WSDL files with the intent of generating code. In particular, this attribute does not provide the name of the web service, which is obtained instead from the <service> element. ***targetNamespace: String (optional)*** This attribute is similar to the one used in XML Schemas. The "targetNamespace" attribute is a convention of XML Schema that enables the WSDL document to refer to itself. The value of this attribute is a URI that becomes the XML namespace for the elements used to describe the services, ports, messages, and bindings defined in the file. It is not necessary (or possible) to explicitly state the namespace when declaring these objects, because they will automatically be associated with the target namespace.

Continued on next page

Table 5.1 – continued from previous page

Associations	***Import (zero or more)*** The <import> element(s) contained in the <definitions> element. ***Types (zero or one)*** The optional <types> element declared in a WSDL document and contained in the <definitions> element. ***Message (zero or more)*** The <message> element(s) declared in a WSDL document and contained in the <definitions> element. ***PortType (zero or more)*** The <portType> element(s) declared in a WSDL document and contained in the <definitions> element. ***Binding (zero or more)*** The <binding> element(s) declared in a WSDL document and contained in the <definitions> element. ***Service (zero or more)*** The <service> element(s) declared in a WSDL document and contained in the <definitions> element. ***Schema (zero or more)*** The schema(s) used in the WSDL document.

Class Name	**Schema**.
Semantics	Represents an external imported XML schema declaration to be used inside a WSDL document. The elements of a schema are used when defining a service, for example when defining the parameters of the operations.
Extends	None.
Attributes	***prefix: String (required)*** The prefix used when referencing the schema. ***namespace: String (required)*** The URI representing the "targetNamespace" attribute of a schema.

Continued on next page

Table 5.1 – continued from previous page

	schemaLocation: String (optional) When importing a schema as an external document the "schemaLocation" attribute is used to indicate the location, in the local file system, of the file containing the schema. When the attribute has a value then the schema is imported inside the <types> element. If it is omitted it means that the schema is accessible over the Internet, and it is declared under the <definitions> element.
Associations	***Element (zero or more)*** The XML schema "element" element(s) declared inside the schema. ***Type (zero or more)*** The XML schema "type" element(s) declared inside the schema.

Class Name	**Element**.
Semantics	Represents the "element" element in an XML schema document. The "complexType" and "simpleType" elements define data types, not actual data elements. The distinction between these two is analogous to the difference between a class and an instance of that class. The data elements are defined using the "element" element.
Extends	None.
Attributes	***name: String (required)*** The name of the element.
Associations	None.

Class Name	**Type**.
Semantics	Represents the "type" element in an XML schema document.
Extends	None.
Attributes	***name: String (required)*** The name of the type.
Associations	None.

Class Name	**Service**.
Semantics	A <service> element groups together a set of related ports. A WSDL document may contain several <service> elements, which are distinguished from each other by their "name" attributes.
Extends	***ExtensibleDocumented***.

Continued on next page

Table 5.1 – continued from previous page

Attributes	***name: String (required)*** The “name” attribute provides a unique name among all services defined within in the enclosing WSDL document.
Associations	***Port (zero or more)*** The <port> element(s) contained in the <service> element. If a service has several ports that share a port type, but employ different bindings or addresses, the ports are alternatives.

Class Name	**Operation**
Semantics	Represents an operation of a Web Service. An operation may have input (zero or one), output (zero or one), and fault (any number) messages. An input message describes the type of message (*e.g.*, SOAP) a client should send to the Web Service. An output message describes the type of message (*e.g.*, SOAP) a client should expect to get back. A fault message describes any error messages (*e.g.*, SOAP) that the Web Service might send back to the client. A fault message is similar to a Java exception. WSDL supports four styles of Web Service messaging: one-way (the operation contains a single input but no output or fault messages, *e.g.*, the client sends a message to the server, to which there is no reply), request-response (the operation contains a single input, followed by a single output, followed by zero or more fault elements, *e.g.*, the operation consists of a message sent from the client to the server, followed by either a response message from the server or a message that reports one of several possible error conditions), solicit-response (the operation contains a single output followed by a single input element, followed by zero or more fault elements, *e.g.*, same as request-response, except that the server sends the first message to the client, thus reversing their roles) and notification (the operation contains a single output but no input or fault elements, *e.g.*, a message sent from the server to the client, to which there is no reply. Such an operation might be used to report an event within the server that the client might need to be aware of).
Extends	***ExtensibleDocumented***.
Attributes	***name: String (required)*** The name of the operation. The value of the attribute is required to be unique within its enclosing <portType> element.

Continued on next page

Table 5.1 – continued from previous page

	parameterOrder: String (optional) Operations do not specify whether they are to be used with RPC-like bindings or not. However, when using an operation with an RPC-binding, it is useful to be able to capture the original RPC function signature. For this reason, an operation may specify an order of parameter names via the "parameterOrder" attribute. The value of the attribute is a list of message part names separated by a single space. The value of the "parameterOrder" attribute must follow specific rules described in the WSDL 1.1 specification. Note that this information serves as a "hint" and may safely be ignored by those not concerned with RPC signatures. Also, it is not required to be present, even if the operation is to be used with an RPC-like binding.
Associations	***Param-Input (zero or one)*** The optional input of an operation. The input must reference exactly one <message> element and has an optional attribute "name: String". The "name" attribute provides a unique name among all input elements within the enclosing port type. ***Param-Output (zero or one)*** The optional output of an operation. The output must reference exactly one <message> element and has an optional attribute "name: String". The "name" attribute provides a unique name among all output elements within the enclosing port type. ***Fault (zero or more)*** The fault element(s) of an operation. Note that only a service-defined exception can be listed as a fault element. The fault element must reference exactly one <message> element and has a required attribute "name: String". Each fault element must be named to allow a binding to specify the concrete format of the fault message. The name of the fault element is unique within the set of faults defined for the operation.

Continued on next page

Table 5.1 – continued from previous page

Class Name	**Import**.
Semantics	WSDL allows associating a namespace with a document location using an import statement. The <import> element allows the separation of the different elements of a service definition into independent documents, which can then be imported as needed. Use of this technique is recommended, in order to allow different Web Services to share the same data types or to separate the definition of a Web Service and its protocol bindings from the elements that provide the address of a server that offers the service. This technique helps writing clearer service definitions, by separating the definitions according to their level of abstraction. It also maximizes the ability to reuse service definitions of all kinds. As a result, WSDL documents structured in this way are easier to use and maintain.
Extends	***ExtensibleAttributesDocumented***.
Attributes	***namespace: String (required)*** The namespace into which the definitions from the included file are to be imported. The value of this attribute must match the target namespace defined in the imported schema document. ***location: String (required)*** A URI that indicates where the imported definitions will be found. This is usually an absolute URL. For example, it could be a relative filename. The specification requires that published WSDL documents use absolute URIs.
Associations	None.

Class Name	**Types**.
Semantics	The customized data types that are used in the messages exchanged by a web service and its clients are defined using the WSDL <types> element, and are referenced from the <message> elements. It is possible to use type definitions found in external schema documents instead of (or as well as) defining types within the WSDL document itself. The WSDL specification recommends the use of XML schema as the preferred schema language, and existing software tools that parse WSDL currently expect to find XML schema elements here.
Extends	***ExtensibleDocumented***.
Attributes	None.
	Continued on next page

Table 5.1 – continued from previous page

Associations	***Schema (zero or more)*** The contained schema definition(s).

Class Name	**Message**.
Semantics	The messages that the service expects to receive or send to its clients. The <message> elements describe the data that is exchanged between the Web Service and its clients in terms of the data types defined within the type elements. In concrete terms, each message defined here corresponds to a SOAP message when SOAP is used as the underlying communications mechanism. A message consists of logical parts, each of which is associated with a definition within some type system.
Extends	***ExtensibleDocumented***.
Attributes	***name: String (required)*** The attribute provides a unique name among all messages defined within the enclosing WSDL document.
Associations	***Part (zero or more)*** The logical part(s) composing the message.

Class Name	**Part**.
Semantics	An item of data that is part of the message. Usually, a single <part> element is used for each method call parameter or return value. A binding may reference the name of a part in order to specify binding-specific information about the part. In general, the data type associated with a part is declared using either a type or an element attribute, only one of which may be specified. However, in the WSDL specification, it is suggested that only element attributes should be used.
Extends	***ExtensibleAttributesDocumented***.
Attributes	***name: String (required)*** The attribute provides a unique name among all the parts of the enclosing message.
Associations	***Element (zero or one)*** The data type associated with the <part> element using the "element" attribute.

Continued on next page

Table 5.1 – continued from previous page

	Type (zero or one) The data type associated with the <part> element using the "type" attribute (*e.g.*, xsd: int). It can be either an XML schema "simpleType" or an XML schema "complexType".

Class Name	**PortType**.
Semantics	A port type is a named set of abstract operations and the abstract messages involved. The operations that a web service provides are represented by <operation> elements. These operations are grouped together as child elements of a <portType> element. You can think of a ¡portType¿ as corresponding to the service endpoint interface, and therefore to the Java interface when the service is implemented in Java. An <operation> element is equivalent to a Java method within that interface.
Extends	***ExtensibleAttributesDocumented***.
Attributes	***name: String (required)*** The attribute provides a unique name among all port types defined within in the enclosing WSDL document.
Associations	***Operation (zero or more)*** The abstract operation(s) of the web service declared inside the <portType> element.

Class Name	**Binding**.
Semantics	A binding defines message format and protocol details for operations and messages defined by a particular port type. There may be any number of bindings for a given port type. A <binding> element contains <operation> elements, similar to the <portType> element. In fact, a <binding> is specific to a particular <portType>: Its <operation>, <input>, <output> and <fault> elements describe the implementation details of the corresponding <portType>. Since SOAP is the most commonly used binding for Web Services, the WSDL specification describes a set of elements that can be used to specify a SOAP binding, but recognizes that it may need to be extended to meet future requirements. The specification also defines elements that can be used to bind a Web Service onto HTTP. A binding must specify exactly one protocol and must not specify address information.

Continued on next page

Table 5.1 – continued from previous page

Extends	***ExtensibleDocumented***.
Attributes	***name: String (required)*** The attribute provides a unique name among all bindings defined within in the enclosing WSDL document.
Associations	***PortType (exactly one)*** A binding references the port type that it binds using the "type" attribute. ***BindingOperation (zero or more)*** The contained <operation> element(s). A <binding> element contains an <operation> element for each operation in its associated <portType>, and each <input>, <output>, and <fault> element in the port type operation also has a corresponding input, output, or fault element here.

Class Name	**BindingOperation**.
Semantics	Represents an operation defined inside a <binding> element. An operation element within a binding specifies binding information for the operation with the same name within the binding's port type.
Extends	***ExtensibleDocumented***.
Attributes	***name: String (required)*** The name of the operation.
Associations	***BindingOperationMessage-Input (zero or one)*** The optional input of a binding operation. The input has an optional attribute "name: String". ***BindingOperationMessage-Output (zero or one)*** The optional output of a binding operation. The output has an optional attribute "name: String". ***BindingOperationFault (zero or more)*** The fault element(s) of a binding operation. Note that only a service-defined exception can be listed as a fault element. The fault element has a required attribute "name: String". Each fault element must be named to allow a binding to specify the concrete format of the fault message.

Continued on next page

Table 5.1 – continued from previous page

Class Name	**Port**.
Semantics	The port element describes how to locate an instance of the port type. It maps a binding of a port type to a URI (address) that can be used to access it using the protocol associated with the binding. Clients to connect to the web service use port addresses.
Extends	***ExtensibleDocumented***.
Attributes	***name: String (required)*** The attribute provides a unique name among all ports defined within in the enclosing WSDL document.
Associations	***Binding (exactly one)*** A port is associated with a binding via its "binding" attribute. The actual address is specified using an element that is specific to the bindings protocol. Here, the "soap:address" element from the SOAP binding is used to provide the URL at which the service endpoint interface for the port can be accessed.
Class Name	**SOAPBinding**.
Semantics	WSDL includes a binding for SOAP 1.1 endpoints, which supports the specification of protocol specific information such as an indication that a binding is bound to the SOAP 1.1 protocol, a way of specifying an address for a SOAP endpoint, the URI for the SOAPAction HTTP header for the HTTP binding of SOAP, the transport protocol used to carry the SOAP messages, whether each operation is RPC-style or document-style, for each part of the input, output, and fault messages associated with the operation, how they are encoded *etc.*. If there are any parts that appear in an attachment, then the MIME binding is used in conjunction with the SOAP binding to describe the structure of the message.
Extends	***ExtensibilityElement***.
Attributes	***style: StyleChoice (optional)*** This attribute is a default that specifies whether the operations in this binding are RPC-style or document-style. It takes the value "rpc" or "document", as appropriate. Each operation can override this default if necessary. If this attribute is omitted, then the style of each operation is taken to be "document" unless otherwise stated in the <soap:operation> element.

Continued on next page

Table 5.1 – continued from previous page

	transport: String (required) Although HTTP is currently the protocol most commonly used to carry SOAP messages, other protocols such as SMTP or even FTP could also be used. The “transport” attribute supplies a URI that identifies the underlying transport protocol.
Associations	None.
Class Name	**SOAPOperation**.
Semantics	Each <operation> element within a binding normally contains a <soap:operation> element that specifies SOAP-related information relating to that operation.
Extends	***ExtensibilityElement***.
Attributes	***style: StyleChoice (optional)*** The attribute indicates whether the operation is RPC-oriented (messages containing parameters and return values) or document-oriented (message containing document(s)). If the attribute is not specified, it defaults to the value specified in the soap:binding element. If the soap:binding element does not specify a style, it is assumed to be “document”. ***soapAction: String (optional)*** The value of this attribute is a URI that becomes the value of the SOAPAction header for the operation. SOAP over HTTP requires that this header be present, even if the service implementation does not use it. If the service does not make use of SOAPAction, then the value should be supplied as an empty string. For other protocols, this attribute should not be supplied at all.
Associations	None.
Class Name	**SOAPBody**.
Semantics	The soap:body element specifies how the message parts appear inside the SOAP Body element. The parts of a message may either be abstract type definitions, or concrete schema definitions. If abstract definitions, the types are serialized according to some set of rules defined by an encoding style. Each encoding style is identified using a list of URIs, as in the SOAP specification.
Extends	***ExtensibilityElement***.
	Continued on next page

Table 5.1 – continued from previous page

Attributes	***use: UseChoice (optional)*** Indicates whether the message parts are encoded using some encoding rules, or whether the parts define the concrete schema of the message. In conjunction with the optional "encodingStyle" attribute they specify how the types listed for the message parts are to be serialized into the message. If "use" has the value "literal", then the associated data is serialized according to its schema in the <types> section of the WSDL document. The value "encoded" specifies that an encoding scheme or series of encoding schemes whose URIs are given by the "encodingStyle" parameter, are used to serialize the data. Although these attributes partly determine the way in which the parts are represented within the SOAP message, the operation style also affects the final encoding. ***encodingStyle: String (optional)*** A list of URIs, each separated by a single space. The URIs represent encodings used within the message, in order from most restrictive to least restrictive (exactly like the encodingStyle attribute defined in the SOAP specification). ***namespace: String (optional)*** It supplies the URI for the namespace to be applied to XML elements created from this part that do not have an explicit namespace assigned as a result of the encoding in use. It may be omitted if not required.
Associations	***Part (zero or more)*** Indicates which parts appear somewhere within the SOAP Body portion of the message (other parts of a message may appear in other portions of the message such as when SOAP is used in conjunction with the multipart/related MIME binding). If the parts attribute is omitted, then all parts defined by the message are assumed to be included in the SOAP Body portion.
Class Name	**SOAPFault**.
Semantics	The soap:fault element specifies the contents of the contents of the SOAP Fault Details element. It is patterned after the soap:body element. The fault message must have a single part. The use, encodingStyle and namespace attributes are all used in the same way as with soap:body, only style="document" is assumed since faults do not contain parameters.

Continued on next page

Table 5.1 – continued from previous page

Extends	***ExtensibilityElement***.
Attributes	***name: String (required)*** Relates the soap:fault to the wsdl:fault defined for the operation. ***use: UseChoice (optional)*** Same as SOAPBody. ***encodingStyle: String (optional)*** Same as SOAPBody. ***namespace: String (optional)*** Same as SOAPBody.
Associations	None.

Class Name	**SOAPHeader**, **SOAPHeaderFault**.
Semantics	The soap:header and soap:headerfault elements allow header to be defined that is transmitted inside the Header element of the SOAP Envelope. It is patterned after the soap:body element. It is not necessary to exhaustively list all headers that appear in the SOAP Envelope using soap:header. The soap:headerfault elements appear inside the soap:header elements.
Extends	SOAPHeader extends ***ExtensibilityElement***.
Attributes	The ***use***, ***encodingStyle*** and ***namespace*** attributes are all used in the same way as with soap:body, only style="document" is assumed since headers do not contain parameters. Additionally, the "use" attribute is required.
Associations	***Message (exactly one)***, ***Part (exactly one)*** Together, the "message" attribute and the "part" attribute reference the message part that defines the header type.

Class Name	**SOAPAddress**.
Semantics	The SOAP address binding used to give a port an address (a URI). A port using the SOAP binding must specify exactly one address. The URI scheme specified for the address must correspond to the transport specified by the soap:binding.
Extends	***ExtensibilityElement***.
Attributes	***location: String (required)*** The URI.
Associations	None.

Continued on next page

Table 5.1 – continued from previous page

Class Name	**HTTPAddress**.
Semantics	The location attribute that specifies the base URI for the port. The value of the attribute is combined with the values of the location attribute of the http:operation binding element.
Extends	***ExtensibilityElement***.
Attributes	***location: String (required)*** The URI.
Associations	None.

Class Name	**HTTPBinding**.
Semantics	WSDL includes a binding for HTTP 1.1s GET and POST verbs in order to describe the interaction between a Web Browser and a web site. This allows applications other than Web Browsers to interact with the site. The protocol specific information may be specified such as an indication that a binding uses HTTP GET or POST, an address for the port or a relative address for each operation (relative to the base address defined by the port).
Extends	***ExtensibilityElement***.
Attributes	***verb: String (required)*** The value of the required verb attribute indicates the HTTP verb. Common values are GET or POST, but others may be used. Note that HTTP verbs are case sensitive.
Associations	None.

Class Name	**HTTPOperation**.
Semantics	The <http:operation> element contains the location attribute.
Extends	***ExtensibilityElement***.
Attributes	***location: String (required)*** The location attribute specifies a relative URI for the operation. This URI is combined with the URI specified in the http:address element of the port, to form the full URI for the HTTP request. The URI value must be a relative URI.
Associations	None.

Continued on next page

Table 5.1 – continued from previous page

Class Name	**HTTPUrlEncoded**.
Semantics	The http:urlEncoded element indicates that all the message parts are encoded into the HTTP request URI using the standard URI-encoding rules. The names of the parameters correspond to the names of the message parts. This may be used with GET to specify URL encoding, or with POST to specify a FORM-POST.
Extends	None.
Attributes	None.
Associations	None.
Class Name	**HTTPUrlReplacement**.
Semantics	The http:urlReplacement element indicates that all the message parts are encoded into the HTTP request URI using a replacement algorithm.
Extends	None.
Attributes	None.
Associations	None.
Class Name	**MIMEMultipartRelated**.
Semantics	The <mime:multipartRelated> element signals that this binding represents a SOAP with attachments message. The WSDL description for a message that contains one or more attachments consists of a set of <mime:part> elements wrapped in a <mime:multipartRelated> element, where the namespace prefix mime is mapped to the URI http://schemas.xmlsoap.org/wsdl/mime.
Extends	***ExtensibilityElement***.
Attributes	None.
Associations	***MIMEPart (zero or more)*** The mime:part element(s) contained in a mime:multipartRelated element.
Class Name	**MIMEPart**.
Semantics	The mime:part element describes each part of a multipart/related message. MIME elements appear within mime:part to specify the concrete MIME type for the part. If more than one MIME element appears inside a mime:part, they are alternatives.
Extends	None.

Continued on next page

Table 5.1 – continued from previous page

Attributes	***name: String (required)*** The name of the part.
Associations	***MIMEMultipartRelated (zero or more)*** The mime:part may contain mime:multipartRelated elements. ***MIMEContent (zero or more)*** The mime:part element may contain zero or more mime:content elements. ***MIMEXml (zero or more)*** The mime:part element may contain mime:mimeXml elements.

Class Name	**MIMEContent**.
Semantics	To avoid having to define a new element for every MIME format, the mime:content element may be used if there is no additional information to convey about the format other than its MIME type string.
Extends	***ExtensibilityElement***.
Attributes	***type: String (optional)*** The type attribute contains the MIME type string. A type value has two portions, separated by a slash (/), either of which may be a wildcard (*). Not specifying the type attribute indicates that all MIME types are acceptable.
Associations	***MIMEPart (zero or one)*** The "part" attribute is used to specify the name of the message part. If the message has a single part, then the part attribute is optional.

Class Name	**MIMEXml**.
Semantics	To specify XML payloads that are not SOAP compliant (do not have a SOAP Envelope), but do have a particular schema, the mime:mimeXml element may be used to specify that concrete schema.
Extends	***ExtensibilityElement***.
Attributes	None.
Associations	***MIMEPart (zero or one)*** Refers to a message part defining the concrete schema of the root XML element. The attribute may be omitted if the message has only a single part. The part references a concrete schema using the element attribute for simple parts or type attribute for composite parts.

As already mentioned, the semantic metamodel presented in Chapter 4 is shared by both the PIM and the PSM in our framework. As a result, figures 5.4, 5.5 and 5.6 depicting the semantic-specific elements of the PSM are essentially identical with figures 4.2, 4.3 and 4.4 in Chapter 4 respectively. The only difference is that the PSM excludes the elements that describe the clients. The excluded elements are not needed in Web Service descriptions. Since these diagrams are essentially identical the documentation of the elements contained in these diagrams is the same as the one provided in Chapter 4, and as a result, for space efficiency, we will not describe it again here.

Chapter 6

Model Transformation Framework

The model transformation process in our framework is a two-step procedure. The first phase involves the transformation of a PIM model to a PSM model and the second phase involves transforming the generated PSM model to WSDL code. The second phase is considered a model transformation phase since code itself can be treated as a model. The model transformation framework is shown in figure 6.1.

When a service provider wishes to generate a description of his/her service, in order to advertise it in a repository, he/she must have the necessary information to achieve that goal. This information includes the details of the service (such as operations, inputs, outputs, *etc.*), context details (such as location of the service, points of availability, *etc.*), and additional semantics (such as the QoS characteristics of the service (*e.g.*, performance metrics)). In addition to this information, the service provider has our PIM domain model for service-oriented systems, presented in chapters 3 and 4, at his/her disposal to describe the service. Using the PIM, in accordance with the service and context details and the rest of the semantics, the service provider creates a PIM service model describing the service.

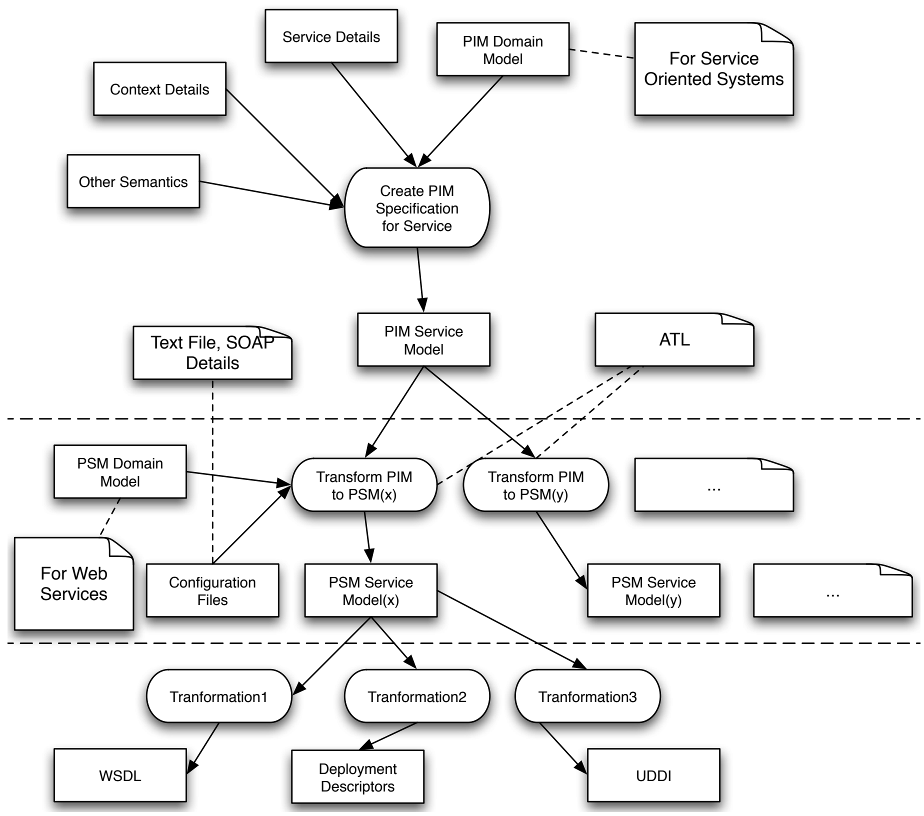


Figure 6.1: Model Transformation Framework

The PIM service model conforms to the PIM domain model. The service provider should also create a configuration file that will be used in the PIM to PSM transformation process. This configuration file contains SOAP specific information, such as the address that the service is accessible to. These are the only tasks a service provider has to perform in our framework. The rest of the tasks are automatically invoked and processed.

The created PIM service model and the configuration file are fed to the PIM to PSM transformation engine. The transformation engine generates the PSM service model corresponding to the PIM service model. The generated PSM models conform to the PSM domain model presented in chapters 4 and 5.

In the next phase the PSM service model is fed into the PSM to extended-WSDL transformation engine. The engine generates the WSDL code corresponding to the PSM service model. Keep in mind that the code is an extended version of WSDL that includes semantic information. The transformation engine could be extended to generate deployment descriptors, UDDI specific files, or any other additional files as needed.

A more elaborate description of the automated tasks following the PIM service model creation is presented in the following paragraphs.

6.1 Phase 1: PIM to PSM

As already mentioned, after the PIM service model and the configuration files are created, they are fed into our PIM to PSM transformation engine. The PIM to PSM transformation engine was implemented using the ATL language[1] and its operations are shown in figure 6.2.

The transformation engine receives as input a PIM service model conforming to our PIM domain model that serves as the metamodel of the service model. All the metamodels used in our transformations conform to the Ecore meta-metamodel. The goal is to transform instances (models) of our PIM domain model to instances (models) of our PSM domain model. The model transformations are performed in accordance to a number of transformation rules. We developed a model describing these transformation rules. The model is presented in the next paragraph. A concrete implementation of the model describing the transformation rules was created using the ATL language. We chose the ATL language to implement our transformation engine simply because it is probably the most

[1] http://www.eclipse.org/m2m/atl/

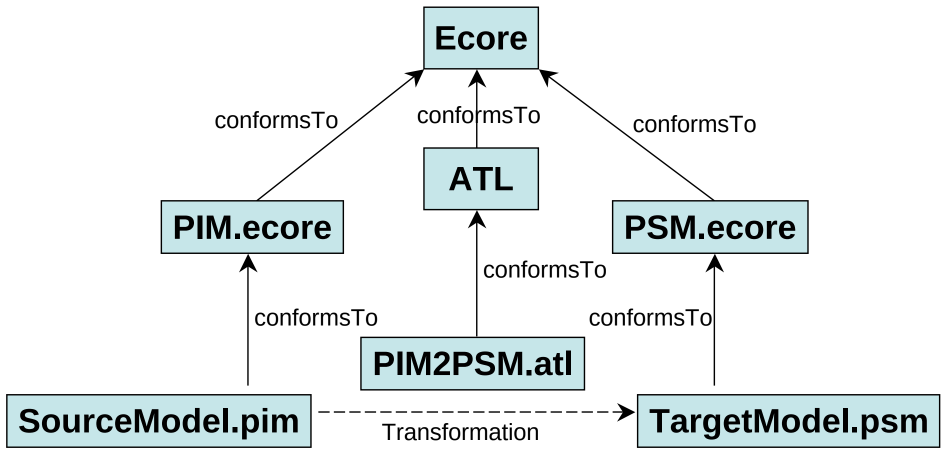


Figure 6.2: PIM to PSM Transformation Engine

widely used free model-to-model transformation tool to date. The ATL language serves as a "transformation" metamodel, in accordance with the PIM and PSM metamodels. The ATL metamodel conforms to the Ecore meta-metamodel as well.

Figure 6.2 introduces the files that will be handled during the execution of the PIM to PSM transformation. These files encode the models (SourceModel.pim, TargetModel.psm), the metamodels (PIM.ecore, PSM.ecore) and the transformation rules (PIM2PSM.atl). The figure presents the transformation of a source file, containing the PIM service model (SourceModel.pim) conforming to PIM.ecore file, to a target file, containing the generated PSM service model conforming to PSM.ecore file. The transformations are performed in accordance with the rules defined in the PIM2PSM.atl file that conforms to the ATL language. The metamodels contained in the files PIM.ecore and PSM.ecore, as well as the ATL language itself, conform to the Ecore meta-metamodel.

At the end of the transformation process the PSM service model will be produced. The generated model contains all the necessary information for generating the WSDL documents. In the next phase the generated PSM service model is fed to the PSM to WSDL transformation engine.

6.1.1 A Model Describing Model Transformations

We will now present the model describing the model transformations taking place in our framework. Due to the fact that both the PIM and the PSM are quite large metamodels, the number of rules describing the transformations between them is quite large as well. This is why we broke down the rules into smaller packages, one package for each rule.

Each package contains on top the rule number, *e.g.*, "R1" means "Rule 1", "R1.1" means the rule executed after "R1" *etc.* Figure 6.3 shows the order in which the rules are executed.

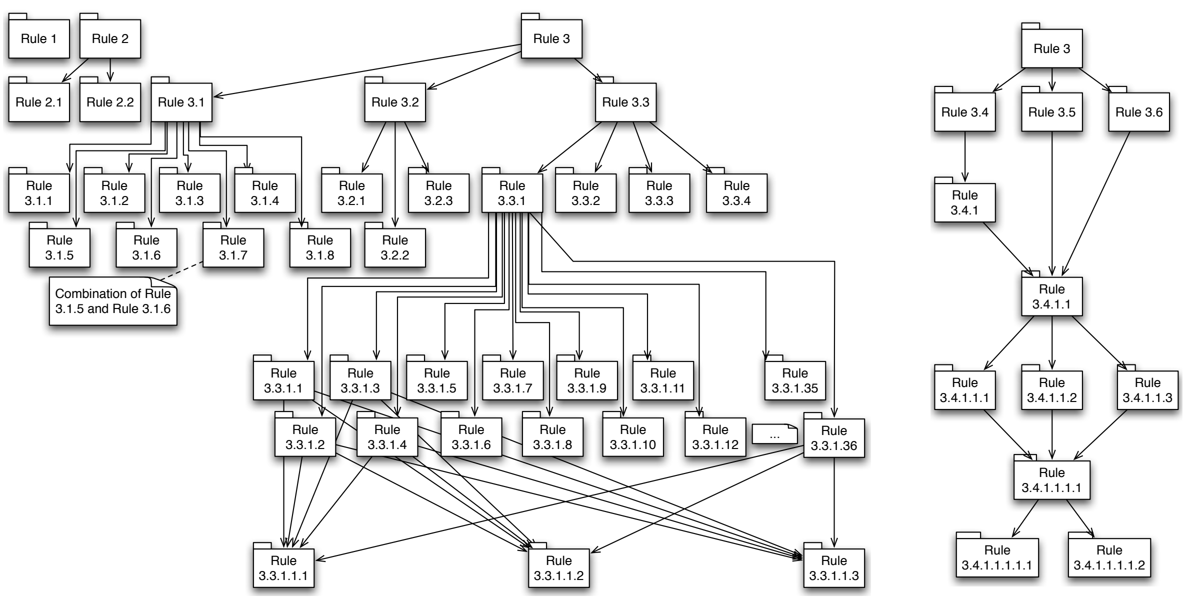


Figure 6.3: Model Describing Model Transformations Part 1

Each package contains the source class(es) of the PIM service model to be transformed, the target class(es) of the PSM service model that are generated, red arrows denoting the source and the generated classes (a red arrow begins from a source class and points at the generated class) and finally black associations that represent the generated associations of the PSM. Each class has as name the metamodel that belongs to plus the name of the class in that metamodel that corresponds to separated by the "!" character. For example, the PIM class "Definitions" would have as its name "PIM!Definitions". In addition, each class contains the attributes that participate in the transformations, either as source or as generated attributes. Consider, for example, in figure 6.4 the "R1" package corresponding to the first rule. The rule describes that the class "Package" of the PIM will be transformed to classes "Definitions" and "Types" of the PSM. In addition, the "targetNamespace:String" attribute of the class "PSM!Definitions" will be generated from the "targetNamespace:String" attribute of the "PIM!Package" class. Furthermore, the containment association between classes "PSM!Definitions" and "PSM!Types" will be generated as well. The same idea is followed with the rest of the transformation rules. We have just described rule R1 above. We will now present the rest of the rules both graphically and by providing a brief description for each one:

- Rule R2: Each "Schema" class of the PIM will be transformed to a class "Schema" of the PSM. In addition, the attributes "prefix:String", "namespace:String" and "schemaLocation:String" of the class "PSM!Schema" will be generated from the "prefix:String", "namespace:String" and "location:String" attributes of the "PIM!Schema" class. Furthermore, the containment association between classes "PSM!Definitions" and "PSM!Schema" will be generated as well.

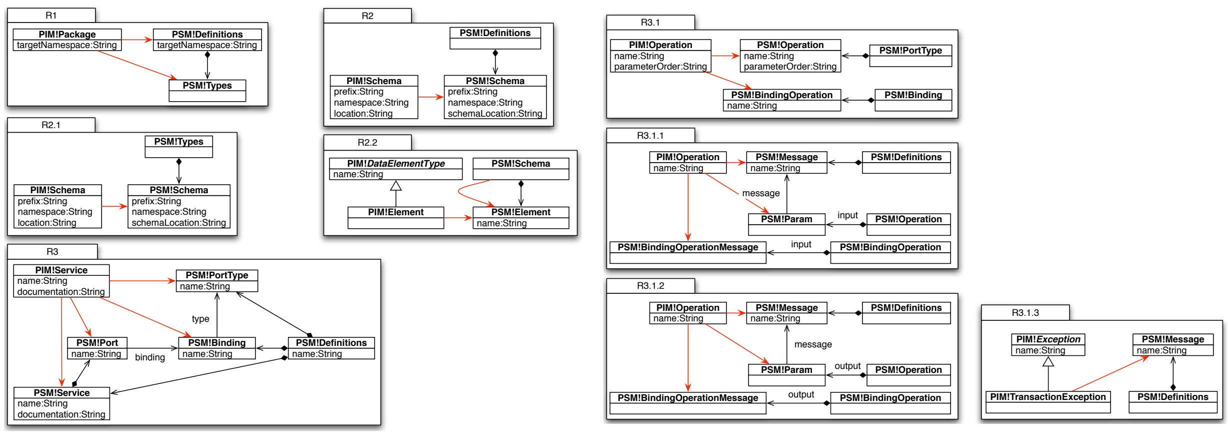


Figure 6.4: Model Describing Model Transformations Part 2

- Rule R2.1: The rule is executed if the "location" attribute of the "PIM!Schema" class has a value. In the same way as in rule R2, the class "PSM!Schema" is generated but this time it is placed under the "PSM!Types" class. The reason is that when a schema is located in an external file (instead of being accessible on the Web), then it is declared inside both the WSDL <Definitions> and <Types> elements.

- Rule R2.2: The rule generates a "PSM!Element" class for each "PIM!Element" class. In addition, the "name:String" attribute of the "PSM!Element" class is generated from the corresponding attribute of the "PIM!Element" class. Furthermore, the "PSM!Element" class is added under the "PSM!Schema" class.

- Rule R3: From the "PIM!Service" class the "PortType", "Port", "Service", and "Binding" classes of the PSM are generated. In addition, the "name:String" attribute of the "PSM!Port", "PSM!PortType", "PSM!Service", "PSM!Definitions", and "PSM!Binding" classes is generated from the "name:String" attribute of the "PIM!Service" class, and the "documentation:String" attribute of the "PSM!Service" class is generated from the corresponding attribute of the "PIM!Service" class. Furthermore, the "PSM!PortType", "PSM!Binding" and "PSM!Service" classes are added under the "PSM!Definitions" class, the "PSM!Port" class is added under the "PSM!Service" class and the "binding" and "type" attributes of classes "PSM!Port" and "PSM!Binding" take the appropriate values.

- Rule R3.1: The rule generates a "PSM!Operation" and a "PSM!BindingOperation" class for each "PIM!Operation" class. In addition, the "name:String" attribute of the generated classes is generated as well from the corresponding attribute of the "PIM!Operation" class. The "parameterOrder:String" attribute of the "PSM!Operation" is generated from the corresponding attribute of the "PIM!Operation". Furthermore, the "PSM!Operation" class is added under the "PSM!PortType", and the "PSM!BindingOperation" is added under the "PSM!Binding" class.

- Rule R3.1.1: The rule generates the request messages for each operation (if applicable). It generates the "PSM!Message", "PSM!Param" and "PSM!BindingOperationMessage" classes from the "PIM!Operation" class. The "name:String" attribute of the "PSM!Message" is generated from the corresponding attribute of the "PIM!Operation" class. The "message" attribute of the "PSM!Param" is generated appropriately, the "PSM!Message" is added under the "PSM!Definitions", the "PSM!Param" is added under the "PSM!Operation" (as input), and the "PSM!BindingOperationMessage" is added under the "PSM!BindingOperation" (as input).

- Rule R3.1.2: The rule generates the response messages for each operation (if applicable). It is almost the same as rule R3.1.1 with the difference that the classes "PSM!Param" and "PSM!BindingOperationMessage" are added as outputs under the classes “PSM!Operation” and “PSM!BindingOperation” correspondingly.

- Rule R3.1.3: The rule generates the exception messages for each operation (if applicable). It generates a “PSM!Message” class for each “PIM!TransactionException” class. The “name:String” attribute of the “PSM!Message” is generated from the corresponding attribute of the “PIM!TransactionException” class. The “PSM!Message” is added under the “PSM!Definitions” class.

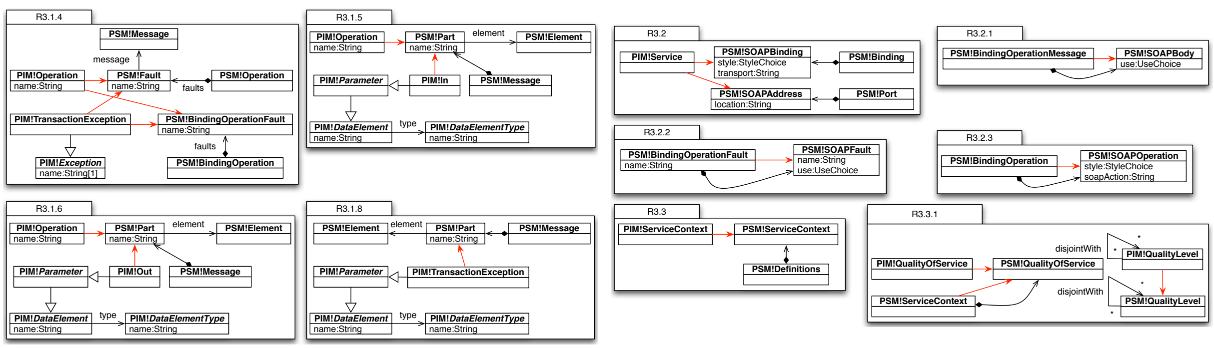


Figure 6.5: Model Describing Model Transformations Part 3

- Rule R3.1.4: The rule completes the generation of the exception messages for each operation (if applicable). It generates the “PSM!Fault” and “PSM!BindingOperationFault” classes from the “PIM!Operation” and “PIM!TransactionException” classes. The “name:String” attribute of the “PSM!Fault” and “PSM!BindingOperationFault” classes is generated from the corresponding attribute of the “PIM!TransactionException” class. The “message” attribute of the “PSM!Fault” is generated appropriately, the “PSM!Fault” is added under the “PSM!Operation” (as fault), and the “PSM!BindingOperationFault” is added under the “PSM!BindingOperation” (as fault).

- Rule R3.1.5: The rule generates the parts of the request messages. It generates the "PSM!Part" class from the "PIM!Operation" and "PIM!In" classes. The "name:String" attribute of the "PSM!Part" is generated from the corresponding attribute of the "PIM!In" class. The "element" attribute of the "PSM!Part" is generated appropriately. The "PSM!Part" is added under the appropriate "PSM!Message" class.

- Rule R3.1.6: The rule generates the parts of the response messages. It generates the "PSM!Part" class from the "PIM!Operation" and "PIM!Out" classes. The "name:String" attribute of the "PSM!Part" is generated from the corresponding attribute of the "PIM!Out" class. The "element" attribute of the "PSM!Part" is generated appropriately. The "PSM!Part" is added under the appropriate "PSM!Message" class.

- Rule R3.1.7: The rule generates the parts of the messages corresponding to the input/output parameters of an operation. It is a combination of rules R3.1.5 and R3.1.6.

- Rule R3.1.8: The rule generates the parts of the exception messages. It generates the "PSM!Part" class from the "PIM!TransactionException" class. The "name:String" attribute of the "PSM!Part" is generated from the corresponding attribute of the "PIM!TransactionException" class. The "element" attribute of the "PSM!Part" is generated appropriately. The "PSM!Part" is added under the appropriate "PSM!Message" class.

- Rule R3.2: The rule and its sub-rules generate the SOAP specific classes. It generates the "PSM!SOAPBinding" and "PSM!SOAPAddress" classes from the "PIM!Service" class. The "style:StyleChoice" and "transport:String" attributes of the "PSM!SOAPBinding" as well as the "location:String" attribute of the "PSM!SOAPAddress" class are generated appropriately receiving default values. The "PSM!SOAPBinding" is added under the "PSM!Binding" and the "PSM!SOAPAddress" is added under the "PSM!Port".

- Rule R3.2.1: The rule generates a "PSM!SOAPBody" class for each "PSM!BindingOperationMessage" class. The "use:UseChoice" attribute of the "PSM!SOAPBody" is generated appropriately receiving a default value. The "PSM!SOAPBody" class is added under the "PSM!BindingOperationMessage".

- Rule R3.2.2: The rule generates a "PSM!SOAPFault" class for each "PSM!BindingOperationFault" class. The "use:UseChoice" attribute of the "PSM!SOAPFault" is generated appropriately receiving a default value and the "name:String" attribute of the "PSM!SOAPFault" class is generated from the corresponding attribute of the "PSM!BindingOperationFault" class. The "PSM!SOAPFault" class is added under the "PSM!BindingOperationFault".

- Rule R3.2.3: The rule generates a "PSM!SOAPOperation" class for each "PSM!BindingOperation" class. The "style:StyleChoice" and the "soapAction:String" attributes of the "PSM!SOAPOperation" are generated appropriately receiving default values. The "PSM!SOAPOperation" class is added under the "PSM!BindingOperation".

- Rule R3.3: The rule generates the "PSM!ServiceContext" class from the "PIM!ServiceContext". The "PSM!ServiceContext" is added under the "PSM!Definitions" class.

- Rule R3.3.1: The rule generates the "PSM!QualityOfService" class from the

"PIM!QualityOfService" and the "PSM!ServiceContext" classes. The "PSM!QualityOfService" is added under the "PSM!ServiceContext". The "disjoint" attribute of the "PSM!QualityOfService" is generated for each "PSM!QualityOfService" class appropriately from the corresponding attribute of the "PIM!QualityOfService" class.

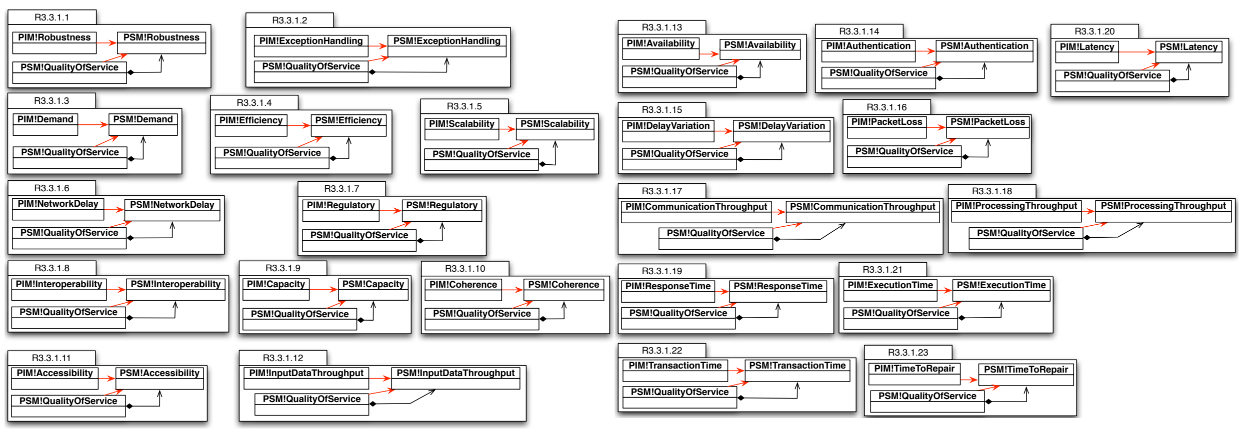


Figure 6.6: Model Describing Model Transformations Part 4

- Rules R3.3.1.1-R3.3.1.36: As already mentioned, the PIM and PSM metamodels of our framework share the semantic part of their definitions that is responsible for describing the semantic information of a service. The shared (common) metamodel was presented in Chapter 4. In order to take advantage of this in our transformations, when we transform the semantic part of our PIM to the corresponding semantic part of the PSM, we essentially copy the semantic part of the PIM to the semantic part of the PSM since they use a common vocabulary. This takes place in rules R3.3.1.1-R3.3.1.36, as well as in the rest of the remaining rules. For example, rule R3.3.1.1 generates a "PSM!Robustness" class for each "PIM!Robustness" class and adds the "PSM!Robustness" class under the "PSM!QualityOfService" class.

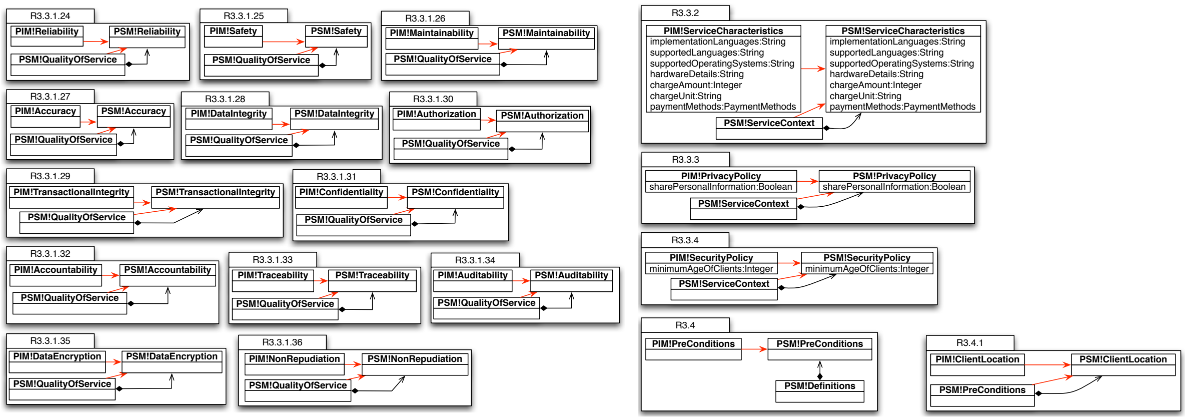


Figure 6.7: Model Describing Model Transformations Part 5

- Rest of the rules: The rest of the rules have as their responsibility to transform (essentially copy) the rest of the semantic descriptions contained in the PIM to the corresponding semantic descriptions of the PSM.

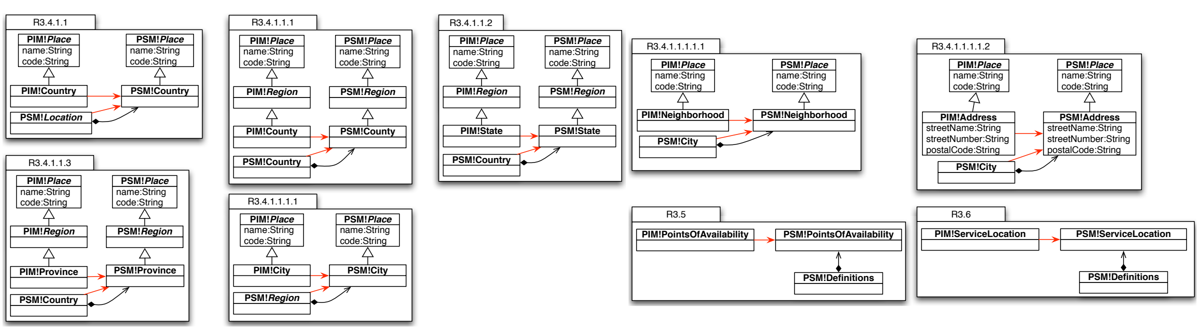


Figure 6.8: Model Describing Model Transformations Part 6

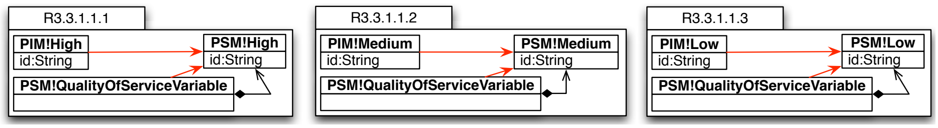


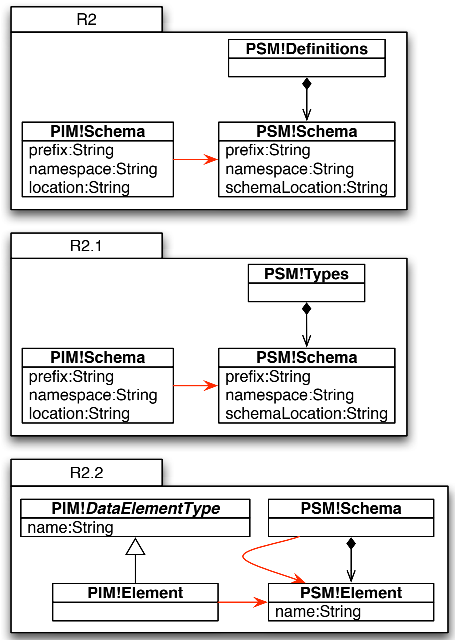
Figure 6.9: Model Describing Model Transformations Part 7

6.1.2 Implementing the Model Describing Model Transformations

We implemented the model describing the model transformations, presented in the previous paragraph, using the ATL language. The ATL is a simple but powerful Java-like transformation language. The implementation of our model using ATL was quite straightforward. The best way to show the simplicity of the ATL language is by providing an example. Consider the rules R2, R2.1 and R2.2 presented in the previous paragraph. The graphical representation of the rules and their ATL implementation is shown in figure 6.10.

As already mentioned, rule R2 (implemented in ATL in rule "SchemaTransformation") transforms each "Schema" class of the PIM to a class "Schema" of the PSM. In addition, the attributes "prefix:String", "namespace:String" and "schemaLocation:String" of the class "PSM!Schema" will be generated from the "prefix:String", "namespace:String" and "location:String" attributes of the "PIM!Schema" class. Furthermore, the "PSM!Schema" class will be placed under the "PSM!Definitions" class.

Rule R2.1 (implemented in ATL in rule "ImportTransformation") is executed if the "location" attribute of the "PIM!Schema" class has a value. In the same way as in rule R2, the class "PSM!Schema" is generated but this time it is placed under the "PSM!Types" class. The reason is that when a schema is located in an external file (instead of being accessible on the Web), then it is declared inside both the WSDL <Definitions> and




```
rule SchemaTransformation {
    from
        schema : PIM!Schema
    to
        schemaPSM : PSM!Schema (
            prefix < − schema.prefix,
            namespace < − schema.namespace,
            schemaLocation < − schema.location
        )
    do {
        if schema.location = ‘’
        then 0
        else thisModule.ImportTransformation(schema)
        endif;
        thisModule.definitionsElement().schemas < − schemaPSM;
        schema.dataElementTypes − > iterate(element; res : Integer = 0
            | thisModule.ElementHandling(element,schemaPSM));
    }
}

lazy rule ImportTransformation {
    from
        schema : PIM!Schema
    to
        schemaPSM : PSM!Schema (
            prefix < − schema.prefix,
            namespace < − schema.namespace,
            schemaLocation < − schema.location
        )
    do {
        thisModule.definitionsElement().types.schemas < − schemaPSM;
    }
}

lazy rule ElementHandling {
    from
        element : PIM!Element,
        schema : PSM!Schema
    to
        elementPSM : PSM!Element (
            name < − element.name
        )
    do {
        schema.elements < − elementPSM;
    }
}
```

Figure 6.10: Implementing the Model Describing Model Transformations

<Types> elements.

Rule R2.2 (implemented in ATL in rule “ElementHandling”) is executed for all the elements declared in a schema. The rule generates a “PSM!Element” class for each “PIM!Element” class. In addition, the “name:String” attribute of the “PSM!Element” class is generated from the corresponding attribute of the “PIM!Element” class. Furthermore, the “PSM!Element” class is added under the appropriate “PSM!Schema” class.

Lazy rules describe sub-rules. Rules R2.1 and R2.2 are sub-rules. In the same way as the one shown in figure 6.10, we implemented the model describing the model transformations, presented in the previous paragraph, using the ATL language.

6.2 Phase 2: PSM to extended WSDL

After the PSM service model is generated by the PIM to PSM transformation engine, a second transformation phase takes place. The transformation takes as input the PSM service model and generates the WSDL code containing all the information specified by the service provider when he/she initially created the PIM service model. All the information of course is taken from the PSM service model. Keep in mind that the WSDL used to describe the Web Services in our framework is an extended version of WSDL. The extensions encompass all the semantic information a Web Service may need in our framework.

Generating source code is a powerful and timesaving procedure, that can help reducing the amount of tedious, redundant, and error-prone programming. However, programs that write code can quickly become very complex and hard to understand. One way to reduce complexity and increase readability is to use templates. The Eclipse Modeling Framework (EMF) project[2] provides a tool for generating source code, called JET (Java Emitter Templates) [44]. With JET we can use a JSP-like syntax (actually a subset of the JSP syntax) that makes it easy to write templates that express the code we want to generate. JET is a generic template engine that can be used to generate any type of source code, including WSDL, from templates. The generated WSDL documents were validated by the Eclipse Web Tools Platform (WTP)[3] and the XMLSpy tool[4].

[2]http://www.eclipse.org/modeling/emf/
[3]http://www.eclipse.org/webtools/main.php
[4]http://www.altova.com/products/xmlspy/xml_editor.html

6.3 A Simple Example

Consider the following example: A "StockQuote" service provides a "GetTradePrices" operation. A "GetTradePrices" SOAP 1.1 request may be sent to the service via the SOAP 1.1 HTTP binding. The operation receives as input a ticker symbol of type string and an application-defined "TimePeriod" structure containing a start and end time, and returns an array of stock prices recorded by the service within that period of time, as well as the frequency at which they were recorded. The RPC signature that corresponds to this service has input parameters "tickerSymbol" and "timePeriod" followed by the output parameter "frequency", and returns an array of floats. The service offers high safety.

A service provider would describe this service in a PIM service model by specifying the following information:

- Specify the namespace of the service description: *e.g.*, http://example.com/stockquote.
- Specify the name of the service: *e.g.*, "StockQuote".
- Add the documentation of the service if necessary: *e.g.*, "A stock quote service".
- Specify schemas as necessary. For each schema:
  - Define the namespace of the schema: *e.g.*, "http://example.com/stockquote/schema".
  - Define the location of the file containing the schema if the schema is available locally: *e.g.*, "StockQuoteSchema.xsd".
  - Specify the schema's elements used in the service definition.
- Define the operations as necessary. For each operation:

  - Define the name of the operation: *e.g.*, “GetTradePrices”.
  - Associate the operation with the service through the “services” attribute.
  - Define the parameters (input, output or exceptions) of the operation. For each parameter:
    * Define the name of the parameter: *e.g.*, “tickerSymbol”.
    * Define the type of the parameter. The type must be included in one of the namespaces defined.
    * Follow the last two steps for each input, output and exception parameter.
  - After defining the parameters of the operation define the “parameterOrder” attribute of the operation if necessary.

- Create a configuration file containing the SOAP address at which the service will be accessible: *e.g.*, “http://localhost:9080/services/StockQuote”.
- Specify the semantic information: *e.g.*, High Safety.

After the service provider finishes the service description he/she may invoke the PIM to WSDL transformation. The transformation will execute initially the PIM to PSM transformation generating the PSM service model and then it will execute the PSM to WSDL transformation generating the WSDL code. Figures 6.11 and 6.12 show the PIM service model describing the “StockQuote” service and the generated PSM service model.

The generated WSDL code is separated in three files: a file containing the abstract definitions of the Web Service, a file containing the concrete specifications and specific service bindings for the Web Service and a file containing the semantic information of the

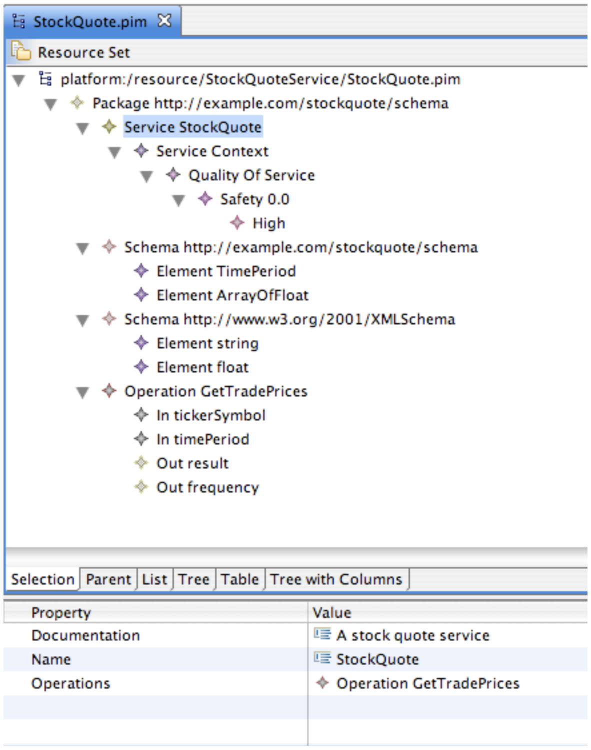


Figure 6.11: PIM Service Model

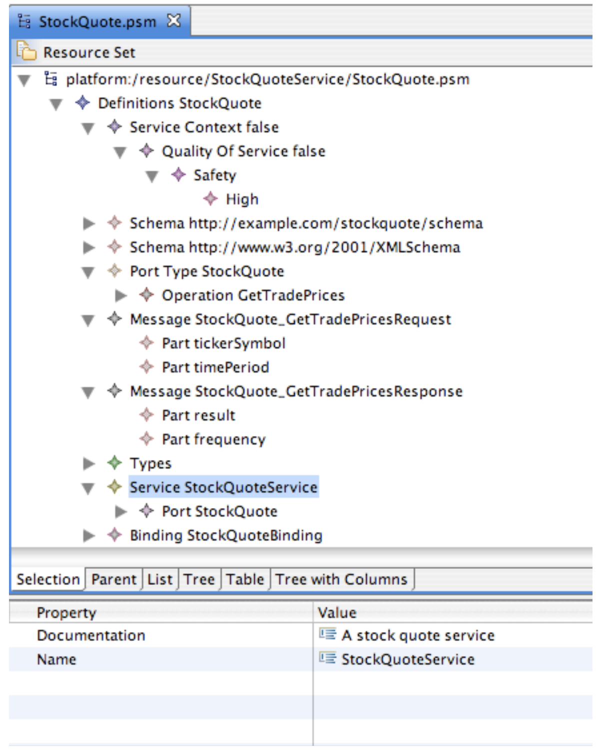


Figure 6.12: PSM Service Model

Web Service. Figures 6.13, 6.14 and 6.15 show the generated WSDL documents containing the description of the service.

StockQuoteInterface.wsdl

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

<wsdl:definitions
   name="StockQuoteInterface"
   targetNamespace="http://example.com/stockquote/schema/interface"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:tns="http://example.com/stockquote/schema/interface"
   xmlns:custom_xsd="http://example.com/stockquote/schema"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <wsdl:types>
      <xsd:schema targetNamespace="http://local/types" xmlns:xsd="http://
www.w3.org/2001/XMLSchema">
                <xsd:import namespace="http://example.com/stockquote/
schema" schemaLocation="StockQuoteSchema.xsd"/>
      </xsd:schema>
   </wsdl:types>

   <wsdl:message name="StockQuote_GetTradePricesRequest">
      <wsdl:part name="tickerSymbol" element="xsd:string"/>
      <wsdl:part name="timePeriod" element="custom_xsd:TimePeriod"/>
   </wsdl:message>

   <wsdl:message name="StockQuote_GetTradePricesResponse">
      <wsdl:part name="result" element="custom_xsd:ArrayOfFloat"/>
      <wsdl:part name="frequency" element="xsd:float"/>
   </wsdl:message>

   <wsdl:portType name="StockQuote">
      <wsdl:operation name="GetTradePrices" parameterOrder="tickerSymbol
timePeriod frequency">
         <wsdl:input message="tns:StockQuote_GetTradePricesRequest"/>
         <wsdl:output message="tns:StockQuote_GetTradePricesResponse"/>
      </wsdl:operation>
   </wsdl:portType>
</wsdl:definitions>
```

Figure 6.13: Abstract Definitions File

StockQuoteService.wsdl

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

<wsdl:definitions
   name="StockQuoteService"
   targetNamespace="http://example.com/stockquote/schema/service"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:tns="http://example.com/stockquote/schema/service"
   xmlns:interface="http://example.com/stockquote/schema/interface">

   <wsdl:import namespace="http://example.com/stockquote/schema/interface"
location="StockQuoteInterface.wsdl"/>

   <wsdl:binding name="StockQuoteBinding" type="interface:StockQuote">
      <soap:binding style="document" transport="http://schemas.xmlsoap.org/
soap/http"/>

      <wsdl:operation name="GetTradePrices">
         <soap:operation style="document" soapAction="http://"/>
            <wsdl:input>
               <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
               <soap:body use="literal"/>
            </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>

   <wsdl:service name="StockQuoteService">
      <wsdl:documentation>A stock quote service</wsdl:documentation>
      <wsdl:port name="StockQuote" binding="tns:StockQuoteBinding">
         <soap:address location="http://localhost:9080/services/
StockQuote"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

Figure 6.14: Concrete Definitions File

StockQuoteContext.wsdl

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

<wsdl:definitions
   name="StockQuoteContext"
   targetNamespace="http://example.com/stockquote/schema/context"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:context="http://schemas.xmlsoap.org/wsdl/context/"
   xmlns:tns="http://example.com/stockquote/schema/context"
   xmlns:service="http://example.com/stockquote/schema/service">

   <wsdl:import namespace="http://example.com/stockquote/schema/service"
location="StockQuoteService.wsdl"/>

   <context:serviceContext>
      <context:qualityOfService>
         <context:safety>
            <context:high id="null" disjointWith=""/>
         </context:safety>
      </context:qualityOfService>
   </context:serviceContext>
</wsdl:definitions>
```

Figure 6.15: Semantic Definitions File

Chapter 7

Service Selection Framework

Imagine the case where a client needs to perform a specific task online, such as booking a flight ticket. This task requires more than one Web Services to collaborate in order to process the client's request; a Web Service for checking the availability of the seats in the specific flight, another for booking the seat, another for charging the client's credit card and so on. We can assume that workflow templates exist describing possible scenarios, such as booking a flight ticket. A workflow template is a set of mock-up Web Services and their execution order, describing a specific task. Let's consider the workflow template describing the booking of a flight ticket. The next step, after finding the right workflow template, would be to instantiate it. For each mock-up Web Service of the workflow there may be many implementations available on the Web, possibly from different service providers, having nearly identical syntactic service descriptions. The question that arises is how are we going to choose the best Web Service from the ones available. Thats where the semantic descriptions of the Web Services come into play.

In our framework, each client has a profile describing his/her preferences including the semantic requirements Web Services should satisfy according to the client. In addition, each Web Service, using the extended WSDL descriptions, will have a set of semantics associated with it. What is left is to find an algorithm to match the preferences of the client to the available Web Service implementations. We propose a service-matching framework based on a number of client and service non-functional characteristics.

Since our intention is to propose a framework and not a specific algorithm, we specified policies containing different matching algorithms that may be used for now in our framework. However, the framework is easily extensible to accommodate new policies and algorithms. To make our service selection framework extensible we used the "Factory" design pattern[1] that enables us to specify each policy in a very modular and extensible way. The common algorithm used in our policies is based on the A^* algorithm, presented in [46]. Each policy combines the A^* algorithm with additional algorithms to form interesting and useful selection policies. We have specified three policies. It is a common sense that in order a client to be able to use a Web Service, the Web Service itself should be available in the area where the client is located. The first policy checks the availability of the Web Service in the area where the client is located. If the Web Service is available in the client's area then the A^* algorithm is performed on the preferences of the client, regarding the QoS characteristics the Web Service should satisfy, and the actual QoS characteristics of the Web Service. A client may have an identification number allowing him/her to use Web Services that are not available to the public. As a security requirement, the authority handling the client requests could specify a specific requirement regarding the valid identifications of the clients that would be able to access specific Web Services. The second policy checks the identification number of the client. If a client is verified then the A^*

[1]http://gsraj.tripod.com/design/creational/factory/factory.html

algorithm is performed on the preferences of the client, regarding the QoS characteristics the Web Service should satisfy, and the actual QoS characteristics of the Web Service. When a Web Service charges the clients, every time they use the specific service, it makes sense that the charging amount would differ according to the time of the day the request is performed. The third policy checks the time at the client's location, when the request is submitted, and adjusts the charging amount accordingly. After the amount is calculated, the A* algorithm is performed on the preferences of the client, regarding the QoS characteristics the Web Service should satisfy, and the actual QoS characteristics of the Web Service.

The service selection framework receives as input a client's profile, the descriptions of the candidate services and the policy under which the selection process should be performed, and finds the best Web Service according to the client's profile and returns the optimal QoS configuration of the Web Service, in case there are alternative possible configurations, as well as a relative score to indicate how close the configuration is to the clients preferences. In addition, it returns the rest of the candidate Web Services with their corresponding optimal configurations and scores.

7.1 A* Algorithm

We will describe how we integrated the A* algorithm in our policies and we will provide an example showing the algorithm in practice. A client may specify in his/her profile a number of QoS characteristics that should be satisfied by the candidate Web Services. In the client's profile, each QoS characteristic C_i specified, is associated with a tuple $<q_i, w_i>$ where q_i is the desired quality level (one of "high", "medium", or "low") and w_i is the corresponding weight (any number) representing the degree of importance the QoS characteristic has for the client. Likewise, a service provider, when defining the description of his/her service, may specify a number of QoS characteristics that the Web Service satisfies associating each QoS variable with one or more quality levels (*e.g.*, "high" or "medium", "high" or "low", *etc.*), meaning that whenever the service provider specifies more than one quality levels for a QoS variable, they will be alternatives. Additionally, a service provider may place constraints between QoS variables, by specifying that the quality level of a specific QoS variable is disjoint with the quality level of another QoS variable (*e.g.*, high quality of accuracy may be disjoint with high quality of latency).

The algorithm works as follows:

1. Get the QoS characteristics requested by the client in his/her profile and place them in a list.

2. Sort the list according to the weight of each QoS characteristic (from higher to lower values).

3. Starting from the first node of the list a search tree is constructed. The root of the tree is the "Root" node, which is essentially an empty node. Each level of the tree corresponds to a QoS characteristic. Each node in the tree indicates a quality level of the QoS variable, a path from the root to the node represents the combined quality levels of the service being evaluated, and the depth of the tree equals to the number of QoS variables requested by the client. The tree is organized according to the relative QoS weights in the client's profile. The heavier the weight of a QoS variable is, the closer to the root is placed as a node. The least weighted QoS variable forms the target nodes. Keep in mind, there will probably be missing nodes representing

quality levels that can never be achieved due to the constraints placed by the service provider (using the “disjoint with” attribute described earlier).

4. Each node of the tree contains, among other information, three numbers, namely g(n), h(n) and f(n). g(n) is the accumulated deviation so far when considering up to the nth quality, h(n) is the minimum possible deviation to reach the desired quality levels from the nth to the kth (last) quality and f(n) is the estimated total deviation cost of path when considering up to the nth quality (f(n)=g(n)+h(n)).

5. Create an empty list (called “open” list) and add the starting (“Root”) node to the open list. Repeat the following:

    (a) Find the node with the lowest f(n) inside the open list. We refer to this node as the “current” node. In case of a tie select the node compared last.

    (b) If the current node is a target node (a leaf node of the tree), then the algorithm is completed, else remove the current node from the open list and add the children of the node in the open list calculating the g(n), h(n) and f(n) for each child. The g(n) is calculated as follows: each node contains information regarding the desired quality level requested by the client, and the actual quality level offered by the service. When a client requests a QoS that is not contained in the description of the service, the algorithm assumes that the offered quality level is “low”. For each node, a “cost” is estimated. Each quality level is given a specific value (“High”=3, “Medium”=2 and “Low”=1). The algorithm subtracts the numbers corresponding to the desired quality level and the actual quality level of the service. If, for example, the client requests a “High” quality for a specific characteristic, and the service offers "Low" then the result will be 2. There are three possible subtraction results: 0, 1 or 2. Each result is given a cost. For 0 the cost is 1, for 1 the cost is 5 and for 2 the cost is 9. The g(n) is calculated as the weight given by the user for the specific QoS variable multiplied by the cost. Since g(n) is the accumulated deviation, we add to it the g(n-1) (the g(n) of the "current" node (the parent of child)). The h(n) is calculated as the remaining distance to achieve the goal (the number of the remaining QoS variables to be processed) multiplied by the least weight of these remaining QoS variables. The f(n) is the sum of the g(n) and h(n).

(c) Return to step (a).

The algorithm returns the node that was selected last (a leaf (target node) of the tree) and the f(n) value of that node. By traversing the search tree backwards, we get the configuration of the Web Service that matches better the preferences of the client. Moreover, the f(n) of the last calculated node is the "final score" of the matching process. The Web Service with the lowest "final score" is the best according to the client's preferences. Although the algorithm might seem a bit confusing, it will become clearer with the example presented in the next paragraph.

7.2 An A^* Algorithm Example for Service Selection

Consider a scenario where a client is searching for a Web Service, for example a flight booking service. In the scenario, there are two Web Services available satisfying the syntactic requirements posed by the client. However, one of these Web Services can be selected and that is where the semantic requirements set by the client come into play.

7.2.1 Client Preferences

The client wishes the service to have at least the following QoS characteristics:

- High latency with 0.35 weight.
- High accuracy with 0.4 weight.
- High safety with 0.25 weight.

Keep in mind the "High", "Medium" and "Low" values refer to the offered quality level of each QoS characteristic. The PIM object diagram depicting the preferences of the client is shown in figure 7.1.

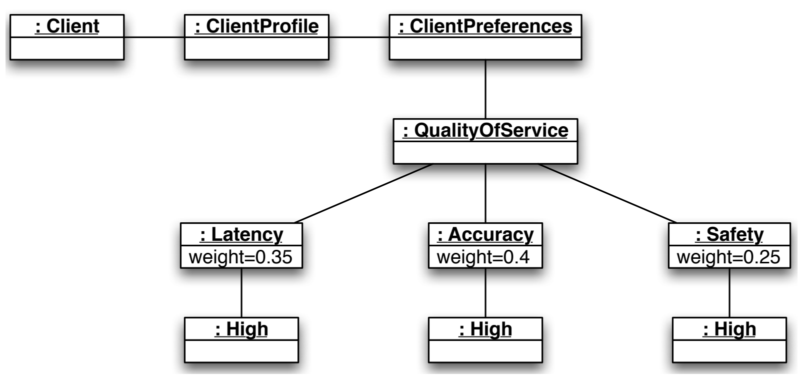


Figure 7.1: A* Example Client Preferences

7.2.2 Web Service Descriptions

Consider the first Web Service matching the syntactic specifications of the request. The service offers high, medium or low latency (as alternatives), high, medium or low accuracy and finally high, medium or low safety. Moreover, the service provider specifies, as constrains, that high latency is disjoint with high safety in his/her domain. Additionally, high accuracy is disjoint with high latency or medium latency. The PIM object diagram depicting the service description is shown in figure 7.2.

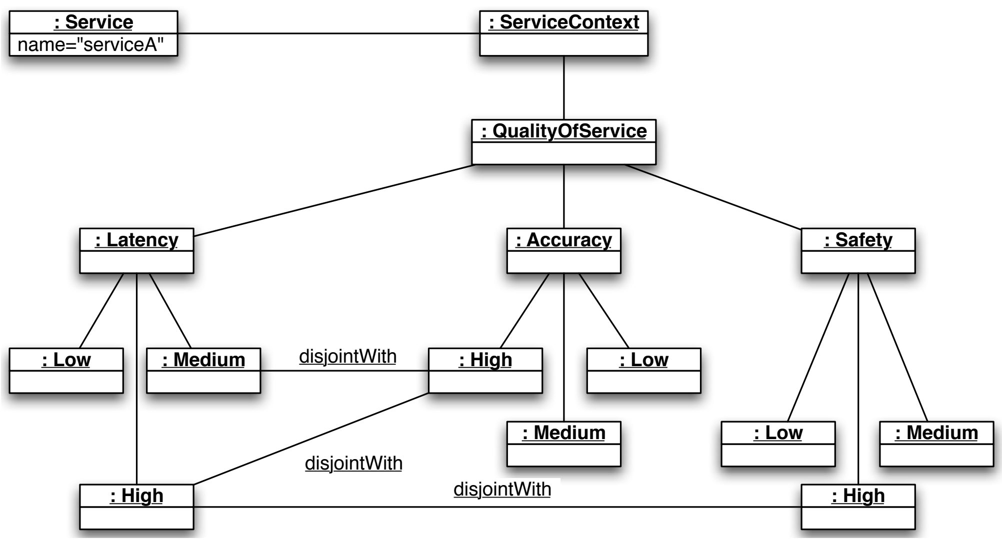


Figure 7.2: A* Example "ServiceA" Description

Consider the second Web Service matching the syntactic specifications of the request. The service offers high, medium or low latency (as alternatives), and high, medium or low accuracy. The description of the service provides no information regarding the "safety" QoS characteristic. In this case, the framework assumes the offered quality level is low. Moreover, the service provider specifies, as constrains, that high accuracy is disjoint with high latency or medium latency in his/her domain. The PIM object diagram depicting the service description is shown in figure 7.3.

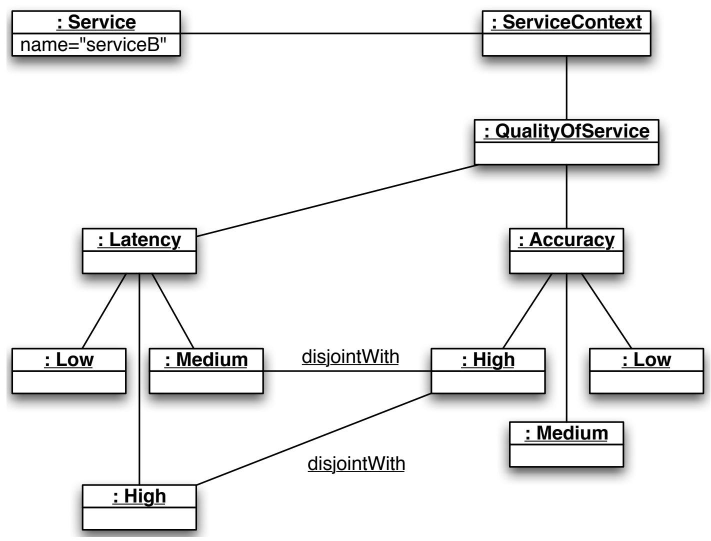


Figure 7.3: A* Example "ServiceB" Description

7.2.3 A* Algorithm Walkthrough

We will describe how the algorithm works when comparing the client's preferences against the semantic information of the first Web Service. The A* algorithm would work as follows:

1. Sort the client's QoS characteristics according to their weight. "Accuracy" would be placed first, "Latency" would be placed second and "Safety" would be placed third.

2. Construct the search tree. The search tree is shown in figure 7.4. Notice there are missing nodes due to the constraints placed by the service provider using the "disjoint with" property.

Let us provide a walkthrough of the algorithm.

1. Create the "open" list and add the "Root" node in the list.

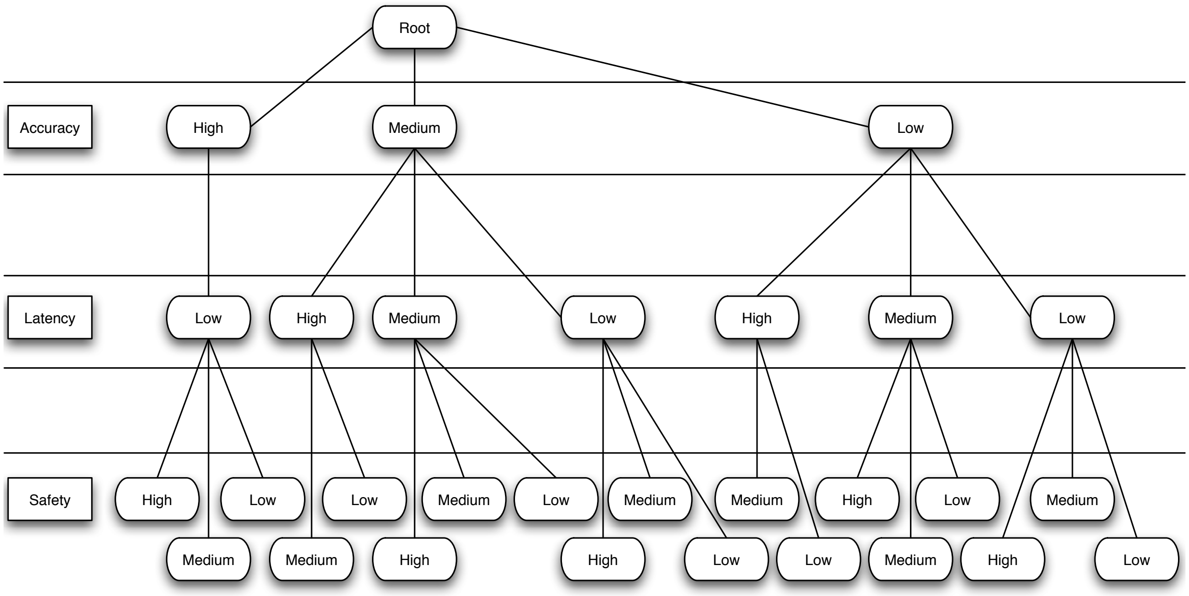


Figure 7.4: A* Example Search Tree

2. The node with the lowest f inside the list (the only node) is the "Root". It is selected and set as the "current" node.

3. Since it is not a target node, the "Root" node is removed from the open list and its children are considered. For each of the children we calculate the g(n), h(n) and f(n) values. The children are placed in the open list. The open list is re-evaluated.

4. The node with the lowest f in the open list is the node "Accuracy High", which is selected and is set as the "current" node. A snapshot of the tree is shown in figure 7.5. The selected node is highlighted in red.

5. Since it is not a target node, the "current" node is removed from the open list and its children are considered. For each of the children we calculate the g(n), h(n) and

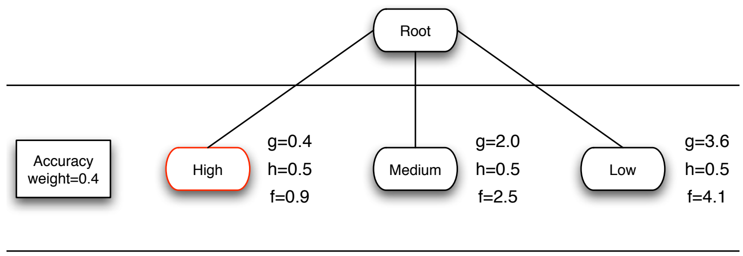


Figure 7.5: A* Search Tree Snapshot 1

f(n) values. The children are placed in the open list. The open list is re-evaluated.

6. The node in the open list with the lowest f is the node "Accuracy Medium". That means that we have to rollback and re-consider the specific node. A snapshot of the tree is shown in figure 7.6. The selected node is highlighted in red.

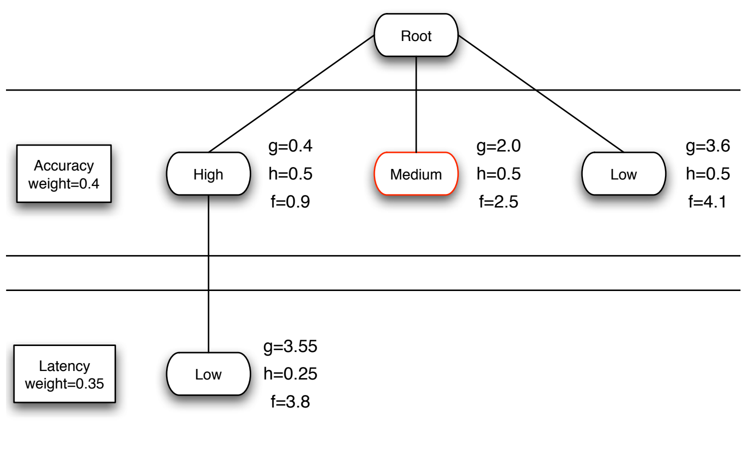


Figure 7.6: A* Search Tree Snapshot 2

7. Since it is not a target node, the "current" node is removed from the open list and its children are considered. For each of the children we calculate the g(n), h(n) and

f(n) values. The children are placed in the open list. The open list is re-evaluated.

8. The node in the open list with the lowest f is the node "Latency High". A snapshot of the tree is shown in figure 7.7. The node is highlighted in red.

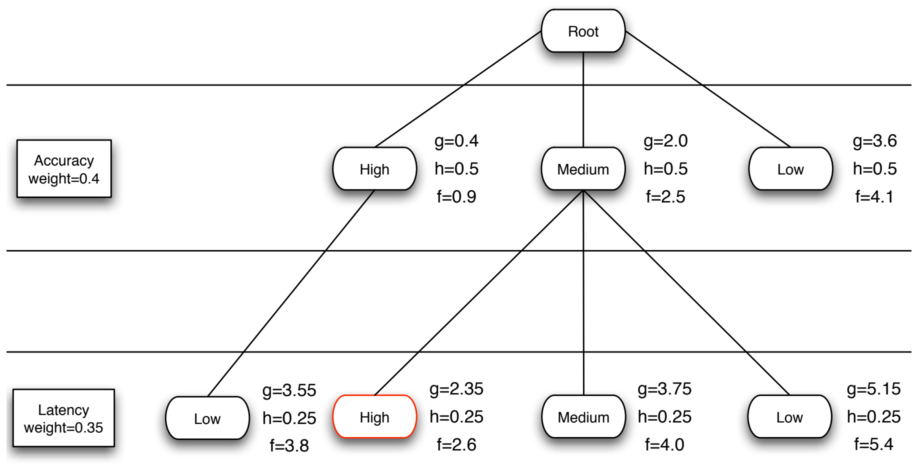


Figure 7.7: A* Search Tree Snapshot 3

9. Since it is not a target node, the "current" node is removed from the open list and its children are considered. For each of the children we calculate the g(n), h(n) and f(n) values. The children are placed in the open list. The open list is re-evaluated.

10. The node in the open list with the lowest f is the node "Safety Medium". A snapshot of the tree is shown in figure 7.8. The node is highlighted in red.

11. Since the node is a leaf of the tree, it is a target node. That means that the algorithm is completed and by traversing the tree backwards we can have the optimal configuration of the Web Service. The f value of the final node is the relativity score of the Web Service. As a result, the optimal configuration would be "Accuracy Medium",

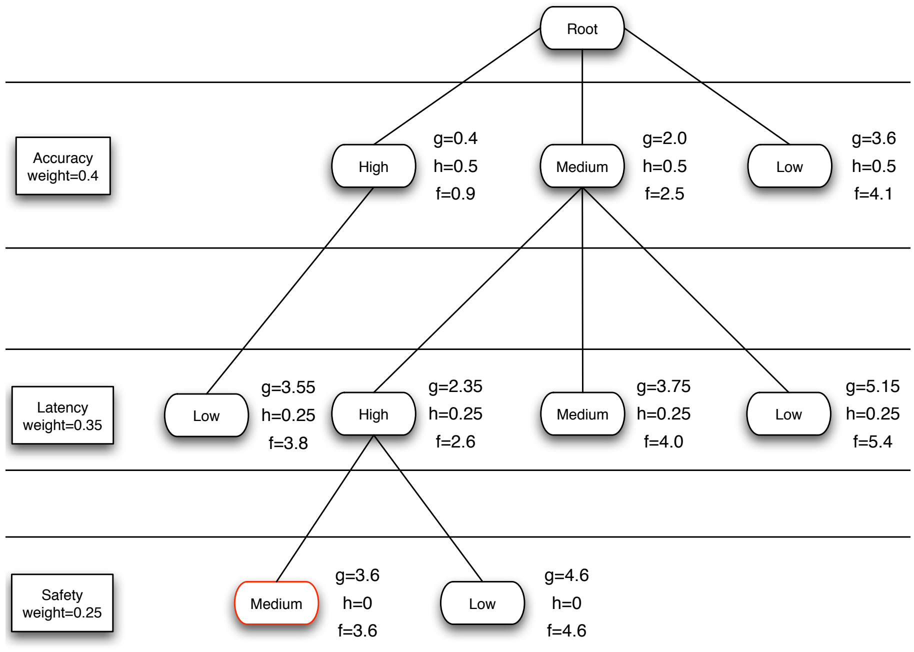


Figure 7.8: A* Search Tree Snapshot 4

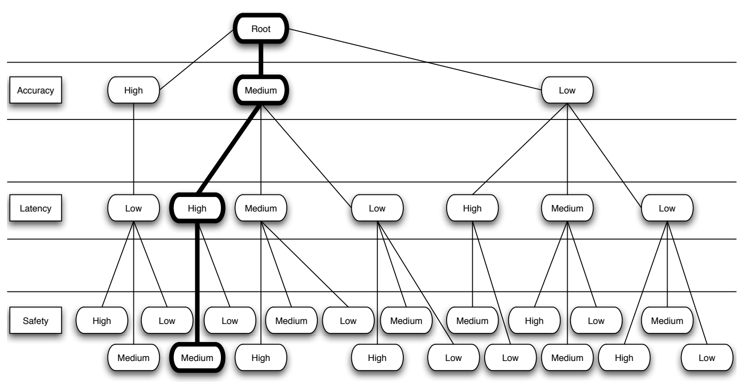


Figure 7.9: A* Search Tree Final Path

"Latency High", and "Safety Medium" with a score of 3.6. The final path of the algorithm is shown in figure 7.9.

By executing the algorithm against the second Web Service description using the same client profile, the algorithm would return as the optimal configuration “Accuracy Medium”, “Latency High”, and “Safety Low” with a score of 4.6. Since the score of “ServiceB” is higher than the score of “ServiceA”, “ServiceA” is more appropriate according to the requirements posed by the specific client. Higher score means greater deviation from the client preferences, thus “ServiceA” was considered more suitable in our example.

7.3 Integrating the Service Selection Framework in Workflows

Consider a workflow template describing a task, such as booking a flight ticket, as mentioned in the introduction of this chapter. Figure 7.10 presents a generic workflow template as well as the integration of the service selection framework in such a workflow.

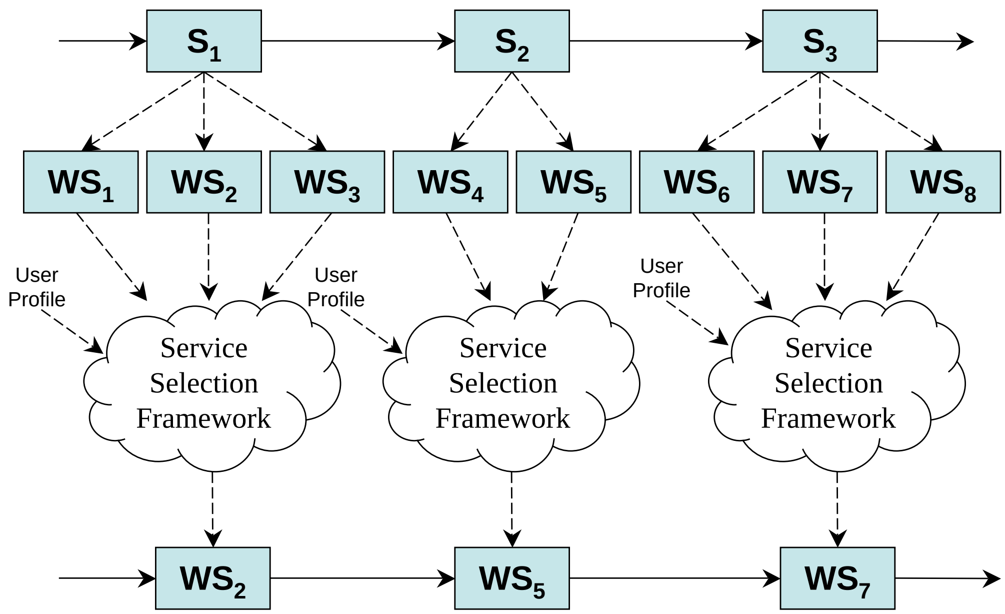


Figure 7.10: The Service Selection Framework in Workflows

The workflow template is composed of three dummy services S_1 (a service for checking the availability of the seats in the specific flight), S_2 (a service for booking the seat) and S_3 (a service for charging the client's credit card). The purpose of this example of course is not to provide a concrete scenario but just a demonstration of how our framework would work with workflows. For service S_1 there are three implementations available WS_1, WS_2

and WS_3. The descriptions of the service implementations are fed in the service selection framework along with the user profile. The service selection framework chooses the best implementation and returns the result. The same procedure is followed for services S_2 and S_3. Now the selected Web Services can be executed using for example BPEL4WS[2].

[2]http://www-128.ibm.com/developerworks/library/specification/ws-bpel/

Chapter 8

A Case Study

In the context of this thesis, we have developed, as proof of concept, a demonstration tool containing everything we have discussed thus far. The tool was implemented on the Eclipse platform as a Rich Client Platform (RCP) application. During the theoretical study, the design and implementation of the tool, a number of technologies were studied and applied:

- Eclipse platform[1]: the Eclipse platform is a "general purpose" IDE (Integrated Development Environment), meaning that the basic version of the platform does not provide any specialized functionality besides being a development platform, however, it provides the foundations and a framework to create plug-ins that can be added to the platform and extend its functionality.

- Rich Client Platform (RCP) technology[2]: RCP is the minimal set of Eclipse plug-ins needed to build a platform application with a UI. RCP tools are implemented inside the Eclipse platform but are capable of running outside the Eclipse platform

[1]http://www.eclipse.org/
[2]http://www.eclipse.org/home/categories/rcp.php

as standalone applications.

- Eclipse Modeling Framework (EMF)[3]: EMF is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor.

- ATL language[4]: the ATL project aims at providing a set of model-to-model transformation tools.

- Java Emitter Templates (JET)[5]: JET enables us to use a JSP-like syntax (actually a subset of the JSP syntax) that makes it easy to write templates that express the code we want to generate.

In the next paragraphs we will present the architecture and the operational profile of the tool.

8.1 Tool's Architecture

Figure 8.1 presents the architecture of our tool. We use a UML component diagram showing the components forming the architecture and the dependencies between them. The components of the architecture are the following:

- PIM: the class interfaces corresponding to our PIM domain model as well as implementations of the interfaces and additional utility classes.

[3]http://www.eclipse.org/modeling/emf/
[4]http://www.eclipse.org/m2m/atl/
[5]http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html

- PSM: the class interfaces corresponding to our PSM domain model as well as implementations of the interfaces and additional utility classes.

- PIM Edit: includes adapters that provide a structured view and perform command-based editing of the PIM model objects.

- PSM Edit: includes adapters that provide a structured view and perform command-based editing of the PSM model objects.

- PIM Editor: the UI for the PIM editor. Essentially it is an EMF tree editor.

- PSM Editor: the UI for the PSM editor. An EMF tree editor for our PSM. The PSM editor is optional since it is not needed to edit a PSM model at all. However, it enables us to create PSM models from scratch if needed (for test purposes), and it allows us to open and view a generated PSM model in an editor, which is more efficient than reading the XMI of the model.

- PIM2PSM: the ATL transformation engine, which is the engine used in our PIM to PSM transformations. The component contains the transformation rules.

- PSM2WSDL: the Java Emitter Templates plug-ins enabling the PSM to WSDL transformations. The component contains the JET templates specifying the transformations.

- Policies and Algorithms: contains the service selection policies and algorithms.

- Eclipse RCP: a minimal set of Eclipse plug-ins providing the essential infrastructures for the tool's operation.

- The dependencies shown in the figure were created after studying each component separately and by revealing the dependencies of each component with the rest of the components of the system, using a feature provided by the Eclipse platform that reveals these dependencies.

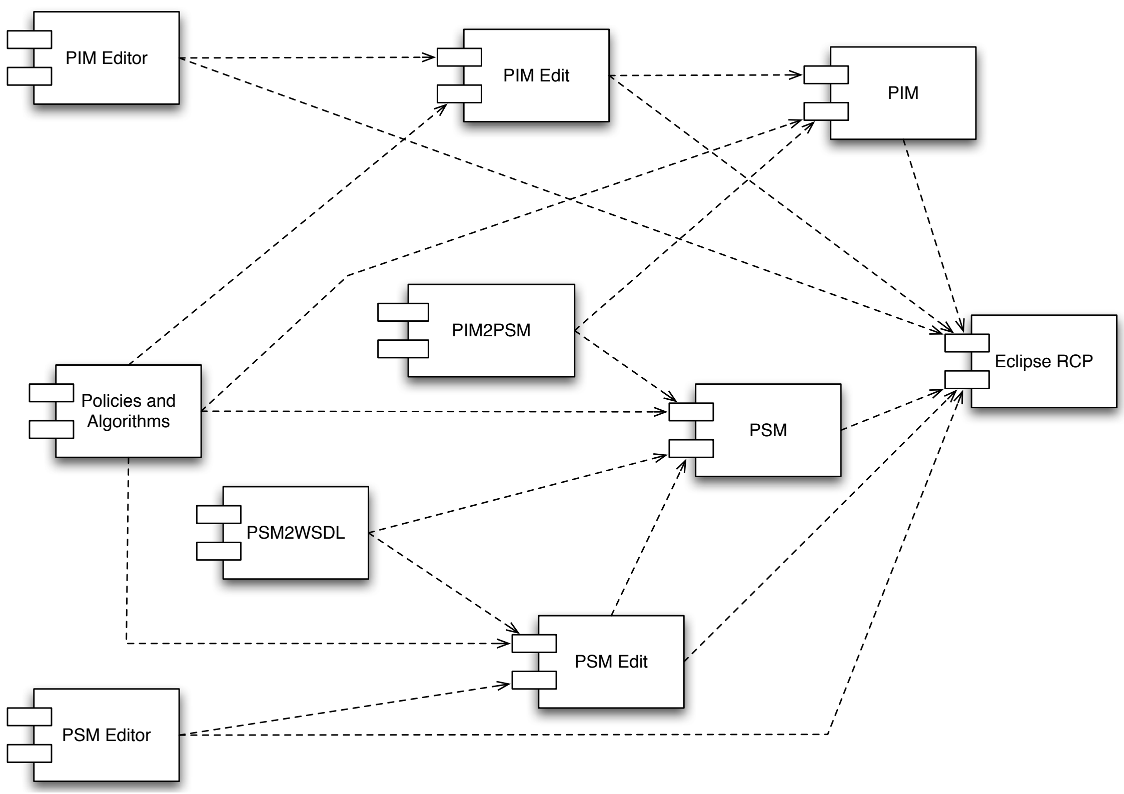


Figure 8.1: Demonstration Tool's Architecture

8.2 Operational Profile

In this section, we will provide a description of the GUI of the tool and a brief description of the features offered by the tool.

8.2.1 Graphical User Interface

A snapshot of the tool is shown in figure 8.2.

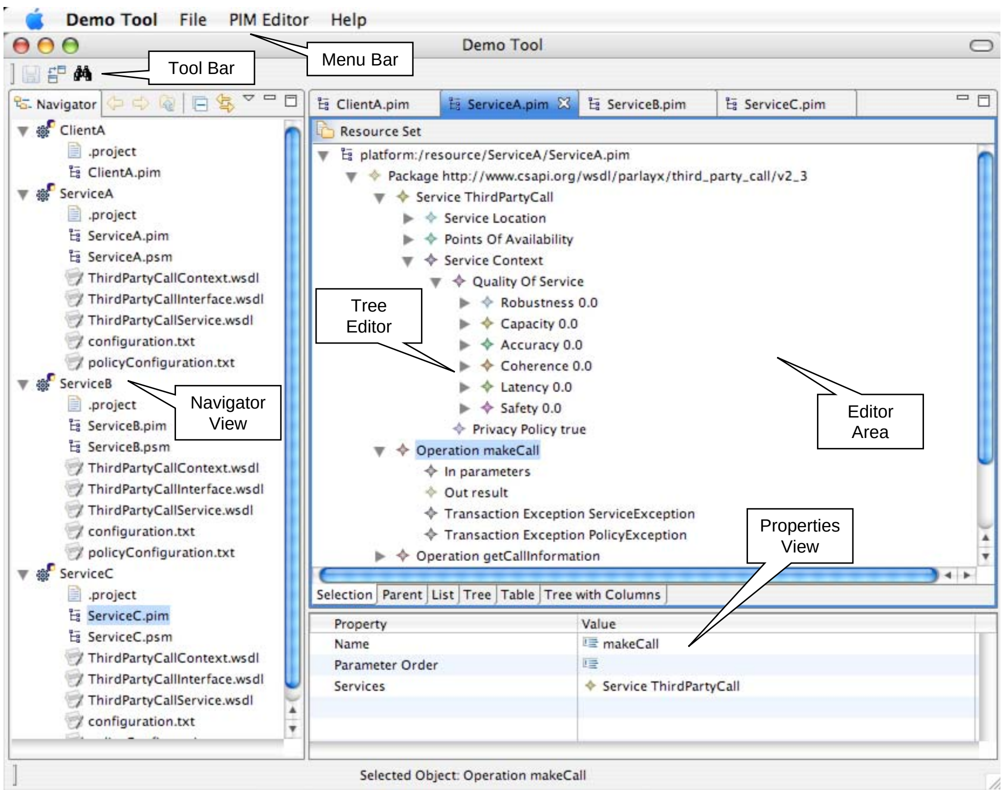


Figure 8.2: Tool Snapshot

As shown in figure 8.2, the tool's GUI is composed of the following parts:

- Menu Bar: the tool has three main menu tabs, "File", "PIM Editor" or "PSM Editor" depending on which editor is opened and "Help" menu tab.
  - "File" tab contains the following options:
    - Save: save the selected diagram.
    - Reset Perspective: resets the layout of the GUI to its defaults.
    - Transform PIM to WSDL: transforms a PIM model to WSDL source files.
    - Service Selection Demo: a demo function of the service selection process.
    - Exit: exit the tool
  - "PIM Editor" and "PSM Editor" tabs contain the following options:
    - Model editing functions such as "New Child" or "New Sibling" actions, adding new child or sibling elements on selected elements in the tree editor. The available elements presented in each function are calculated dynamically from the PIM or PSM metamodel respectively.
    - Validate: a function validating the current model against the metamodel it conforms to.
    - Refresh: refresh the tree editor.
  - "Help" tab contains the "About" function, showing information related to the tool.
- Tool Bar: the tool bar contains the "Save", "Transform PIM to WSDL" and "Service Selection Demo" functions.
- Navigator View: it shows the current resources (created client and/or service descriptions) and enables the creation, opening and deletion of these resources.

- Editor Area: the area of the GUI where a client or service provider opens and edits his/her descriptions graphically.
- Tree Editor: the tree editor containing the PIM or PSM model descriptions. The tree editor comes with a context menu containing functions such as "New Child", "New Sibling", "Undo", "Redo", "Cut", "Copy", "Paste", "Delete", "Validate" and "Refresh".
- Properties View: it shows the properties of each selected element in the tree editor. Both the PIM and PSM descriptions can be edited using this view.
- The tool contains wizards for creating new PIM and PSM models, as well as activating the "Transform PIM to WSDL" and "Service Selection Demo" functions.

8.3 Carrier Services and a Working Example

8.3.1 Parlay X Web Services Specification

The Parlay X Web Services Specification, Version 3.0[6] is a specification issued by the Parlay Group. The Parlay Group is a multi-vendor consortium formed to develop open, technology-independent application programming interfaces (APIs) that enable the development of applications that operate across multiple, networking-platform environments. The Parlay X 3.0 specification has been defined jointly between the European Telecommunications Standards Institute (ETSI), Parlay, and the Third Generation Partnership Program (3GPP). It consists of twenty Web Services:

[6] http://www.parlay.org/en/specifications/pxws.asp

1. Common: specifies the common aspects of the Parlay X 3 Web Services.

2. Third Party Call: a Web Service for creating and managing a call initiated by an application (third party call). The overall scope of this Web Service is to provide functions to application developers to create a call in a simple way. Using the Third Party Call Web Service, application developers can invoke call handling functions without detailed telecommunication knowledge.

3. Call Notification: a Web Service for handling calls initiated by a subscriber in the network. A (third party) application determines how the call should be treated. The overall scope of this Web Service is to provide simple functions to application developers to determine how a call should be treated. It is possible to request to end the call, continue the call or re-route the call. Optionally, it is also possible to request the media type(s) when the action is to re-route the call.

4. Short Messaging: a Web Service for sending and receiving SMS messages. The overall scope of this Web Service is to provide to application developers primitives to handle SMS in a simple way.

5. Multimedia Messaging: defines a Multimedia Messaging Web Service that can map to SMS, EMS, MMS, IM, E-mail, *etc.*

6. Payment: supports payment reservation, pre-paid payments, and post-paid payments. It supports charging of both volume and currency amounts, a conversion function and a settlement function in case of a financially resolved dispute.

7. Account Management: supports account querying, direct recharging and recharging

through vouchers. As a side effect, it may prevent subscribers from having their account balance credits expire.

8. Terminal Status: provides access to the status of a terminal by requesting for the status of a terminal or requesting for the status of a group of terminals or through the notification of a change in the status of a terminal.

9. Terminal Location: provides access to the location of a terminal by requesting for the location of a terminal, or requesting for the location of a group of terminals, or through the notification of a change in the location of a terminal, or through the notification of terminal location on a periodic basis. The location is expressed through a latitude, longitude, altitude and accuracy.

10. Call Handling: provides a mechanism for an application to specify how calls are to be handled for a specific number. It includes commonly utilized actions such as call accepting (only accepting calls from a list of numbers), call blocking (blocking calls if they are on a blocking list), conditional call forwarding (changing the destination of a call to another number for a specific calling number), unconditional call forwarding (changing the destination of a call to another number), and play audio (initiate audio with the caller (*e.g.*, an announcement or menu)).

11. Audio Call: allows media to be added/dropped for any ongoing call. This Web Service also allows interaction with other call control Web Services (*e.g.*, multimedia conference, third party call), enabling delivery of multimedia to call participants in an ongoing call.

12. Multimedia Conference: a simple Web Service that allows the creation of a multimedia conference and the dynamic management of the participants involved.

13. Address List Management: defines two related interfaces. The first interface manages the groups themselves (creation, deletion, query and access right management). The second interface manages the members within a group, supporting add, delete and query operations. Addresses are not created using this service, they must already exist.

14. Presence: allows for presence information to be obtained about one or more users and to register presence for the same. It is assumed that the typical client of these interfaces is either a supplier or a consumer of the presence information. An Instant Messaging application is a canonical example of such a client of this interface.

15. Message Broadcast: provides operations for sending a broadcast message to the network and a polling mechanism for monitoring the delivery status of a sent broadcast message. It also provides an asynchronous notification mechanism for broadcast delivery status.

16. Geocoding: while the Parlay X Terminal Location Web Service provides access to the geographical coordinates at which a terminal is located, the Geocoding Web Service provides an additional level of refinement, allowing the service developer to work with actual location addresses and the like.

17. Application-driven Quality of Service: enables applications to dynamically change the quality of service (*e.g.*, bandwidth) available on end user network connections. Changes in QoS may be applied on either a temporary basis (*i.e.*, for a defined period

of time), or as the default QoS to be applied for a user each time they connect to the network.

18. Device Capabilities and Configuration: allows applications to get information about device capabilities and push device configuration to a device.

19. Multimedia Streaming Control: controls the consumption of streaming multimedia of a service provided to an end-user.

20. Multimedia Multicast Session Management: allows for a third party (*e.g.*, application) to control a multicast session, its members and multimedia stream, and obtain channel presence information.

8.3.2 A Working Example

To validate our approach, we created a scenario using, in our syntactic side descriptions, one specification from the Parlay X Web Services Specifications, Version 3.0. More specifically, we chose to generate the WSDL descriptions for a "Third Party Call" Web Service. In a nutshell, the "Third Party Call" Web Service specification defines that a Web Service with name "ThirdPartyCall" has seven operations, namely "makeCallSession", "addCallParticipant", "transferCallParticipant", "getCallParticipantInformation", "getCallSessionInformation", "deleteCallParticipant" and "endCallSession". Each operation contains specific inputs, outputs and exceptions. The Web Service uses SOAP as the messaging protocol and HTTP as the transport protocol. The contained operations are document-style (soap:operation style="document"), and the parts of the messages define the concrete schema according to which they are serialized (soap:body use="literal"). Finally, the Web

Service can be accessed using the address "http://localhost:9080/ThirdPartyCallService/services/ThirdPartyCall". In addition to the syntactic description of the Web Service, we added semantic information. This information included the physical location of the Web Service (country=Canada), the points of availability of the Web Service (country=Canada, province=Ontario, city=Kitchener, postal code=N2H2H6), the information that the Web Service would share the personal information of its clients if necessary, and finally, QoS characteristics (Robustness=High, Capacity=Low | Medium, Accuracy=Low | Medium | High, Coherence=Medium, Latency=Low | Medium | High, Safety=Low | Medium | High). In addition, we specified that High Accuracy is disjoint with High Latency or Medium Latency or Medium Capacity, and High Latency is disjoint with High Accuracy or High Safety. A snapshot of the PIM model containing the description of the specific service is shown in figure 8.3.

In the same way, we created two more Web Service descriptions having identical syntactic descriptions but altered semantic descriptions. The second Web Service (Service B) had the same physical location (country=Canada), altered points of availability (country=Canada, province=Ontario, city=Waterloo, postal code=N2L3G1 and N2L3L1 as well as country=Canada, province=Ontario, city=Guelph, postal code=N2L3XX), same privacy policies, and altered QoS characteristics (Robustness=High, Capacity=Low | Medium, Accuracy=Low | Medium | High, Coherence=Medium, Latency=Low | Medium | High). In addition, we specified that High Accuracy is disjoint with High Latency or Medium Latency or Medium Capacity (same as before), and High Latency is disjoint with High Accuracy. The third Web Service (Service C) had the same physical location (country=Canada), altered points of availability (country=Canada, province=Ontario, city=Kitchener, postal code=N2L3G1 and N2H2L4), same privacy policies, and altered QoS characteristics (Ro-

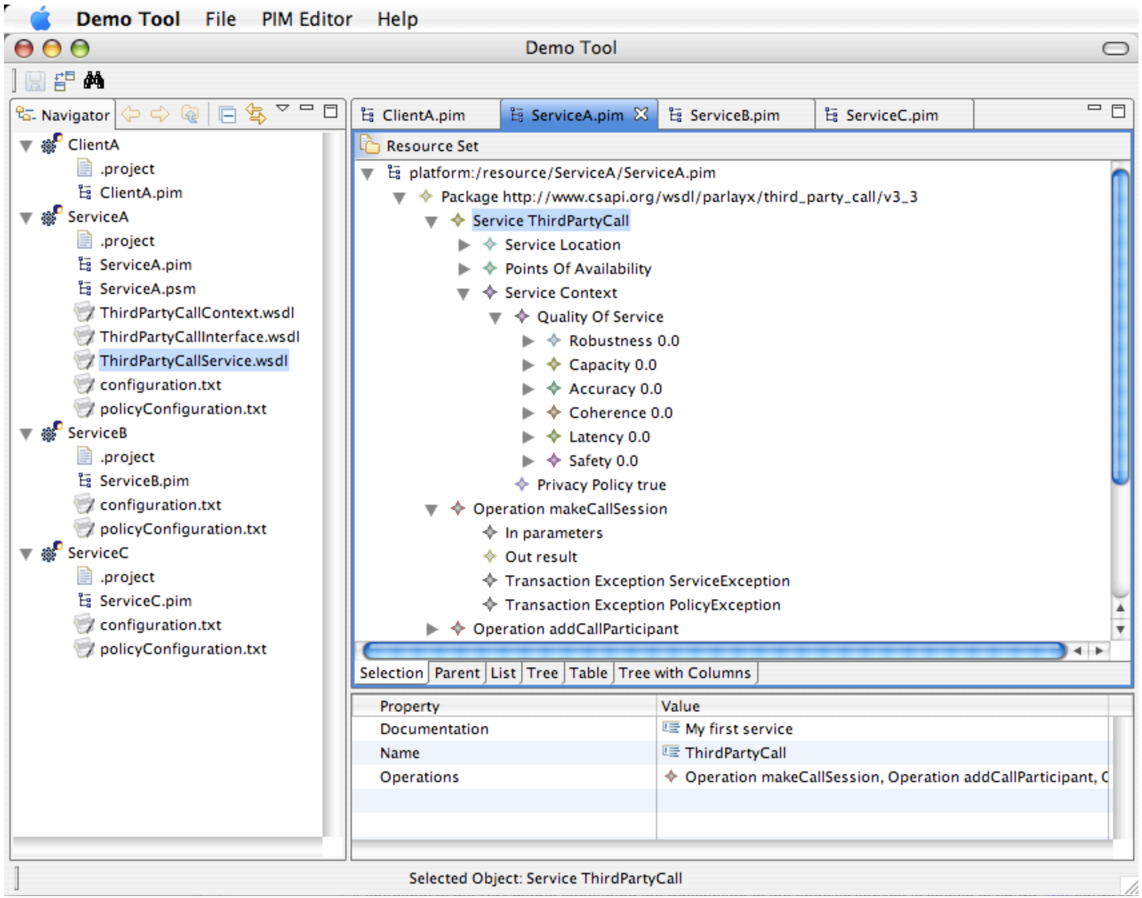


Figure 8.3: Third Party Call PIM Model

bustness=High, Capacity=Low | Medium, Accuracy=Low | High, Coherence=Medium, Latency=Low | Medium | High, Safety=Low | Medium | High). In addition, we specified that High Accuracy is disjoint with High Latency or Medium Latency or Medium Capacity, and High Latency is disjoint with High Accuracy or High Safety (same as Service A).

The next step a service provider should follow after creating a Web Service description is to transform the PIM model containing the Web Service description to its corresponding WSDL source code descriptions. This can be achieved by invoking the "Transform PIM to WSDL" wizard, shown in figure 8.4.

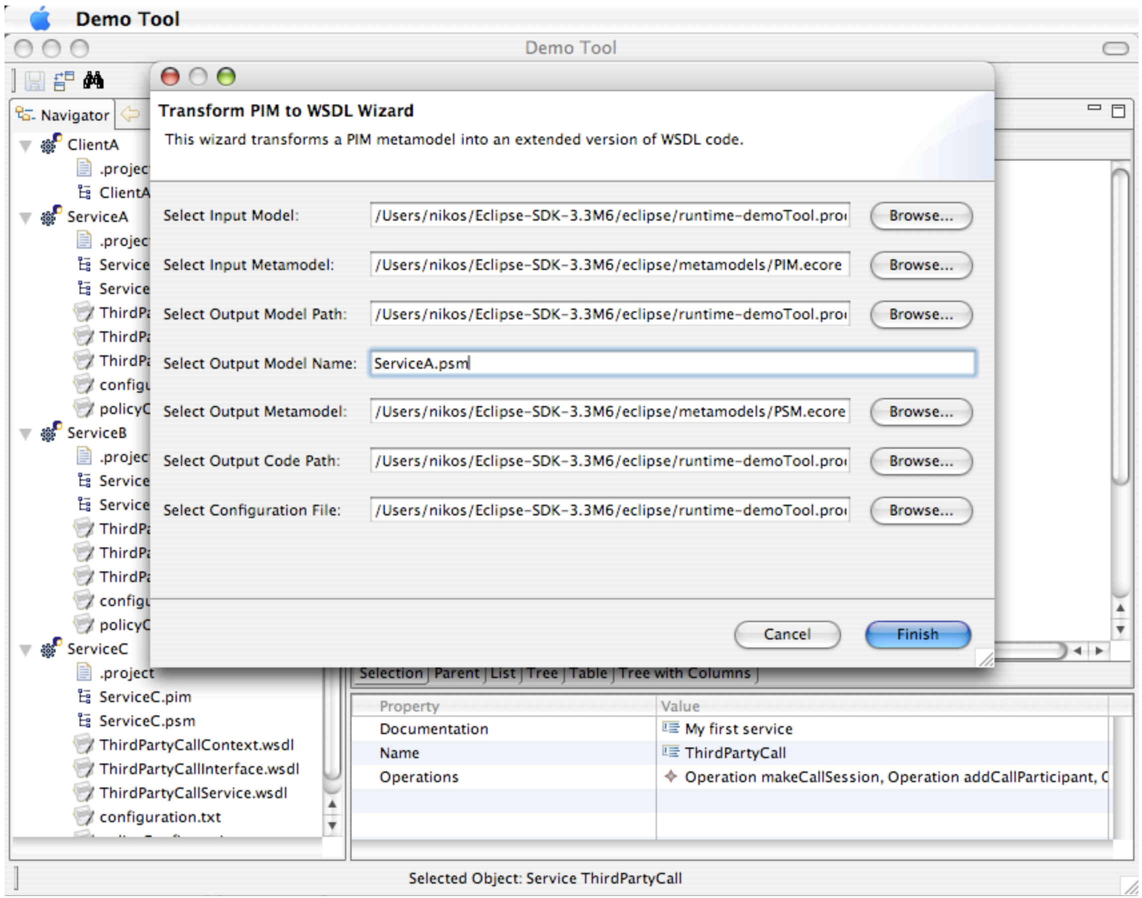


Figure 8.4: Transform PIM to WSDL Wizard

A service provider should additionally create a "configuration.txt" file containing SOAP specific information (the address at which the Web Service can be accessed). This information could not be included in the PIM model because it is technology-specific information. Any technology-specific information required in future releases should be specified inside the configuration file. After invoking, configuring and executing the "Transform PIM to WSDL" wizard, four files are generated:

- The generated PSM model.
- A WSDL file containing the abstract definitions of the Web Service.
- A WSDL file containing the concrete specifications and specific service bindings for the Web Service.
- A WSDL file containing the semantic information of the Web Service.

All WSDL files were validated by the Eclipse Web Tools Platform (WTP)[7] and the XMLSpy tool[8]. It should be noted that all the custom data types used in the Web Service descriptions should be created in separate XML schema documents using one of the plethora of available tools for specifying XML schemas. These data types can be imported in our tool as necessary, but should be defined independently.

We described how a service provider can specify and generate the descriptions of his/her Web Services. A client can specify in the same fashion his/her profile. This specification doesn't include transformations because a client's PIM model contains all the information that is needed when selecting the appropriate Web Services for a specific client in our framework. In our example, we created a client with name "Nikolaos Giannopoulos" (how modest is that...), ID="123456789", and location "country=Canada, province=Ontario, city=Kitchener and postal code=N2H2ZZ". The client specifies that a Web Service should ideally have at least the following QoS characteristics: High Latency (weight 0.35), High Accuracy (weight 0.4) and High Safety (weight 0.25). The weight, as already mentioned, reveals how important a QoS characteristic is for the client. The PIM model containing the profile is shown in figure 8.5.

[7]http://www.eclipse.org/webtools/main.php
[8]http://www.altova.com/products/xmlspy/xml editor.html

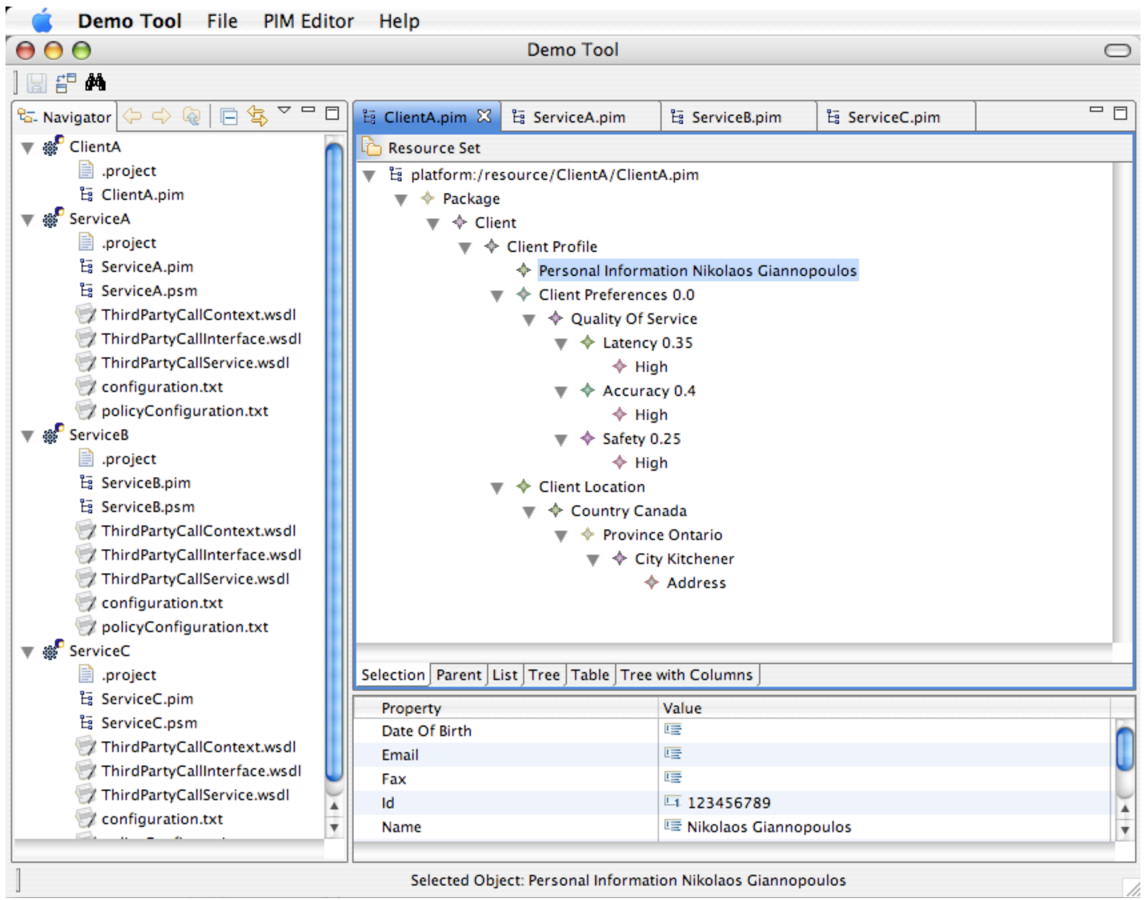


Figure 8.5: A Client's PIM Model

Consider this client is searching to invoke a "Third Party Call" Web Service. In a repository containing services A, B and C, all three of these services would have been candidates for the specific request. However, all three Web Services have identical syntactic descriptions, so the question that arises is which of these services is the best for the specific client? In order to answer that we will use the "Service Selection Demo" wizard demonstrating how our framework would work under these conditions. The wizard is shown in figure 8.6.

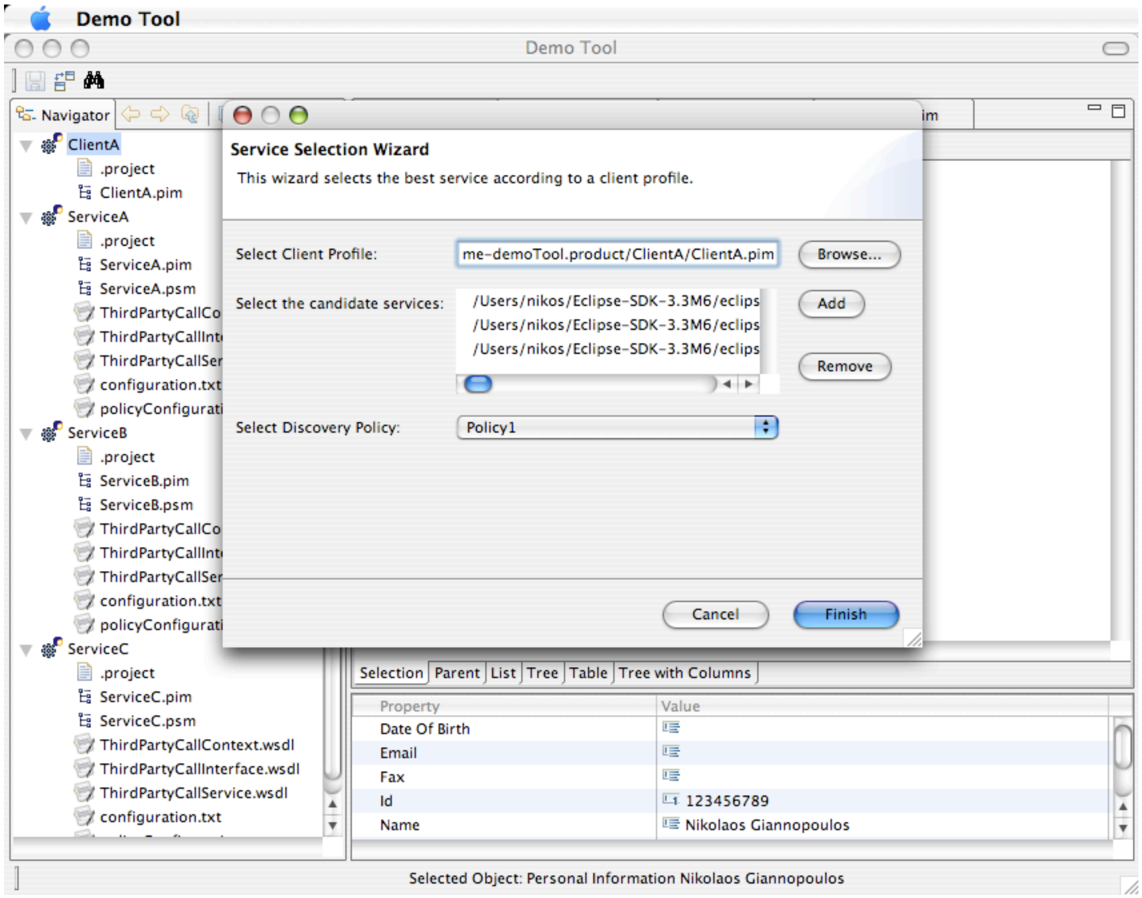


Figure 8.6: Service Selection Demo Wizard

We have already presented the policies that were defined in our framework. The authority handling the client requests should create a policy configuration file (policyConfiguration.txt) that contains the policies each Web Service is compliant with. The "Service Selection Demo" wizard updates dynamically the policies that can be used in the Web Service selection process by reading the policy configuration file and by taking the intersection of the specified policies. It is specified that service A is compliant with Policies 1, 2 and 3, service B is complaint with Policies 1 and 3, and service C is compliant with policies

1 and 2. When checking the client's profile against Web Services A, B and C, only one policy becomes available that is policy 1. As a reminder, policy 1 checks the availability of each Web Service in the client's location and if it is available it is tested against the A* algorithm. The results of the service selection process are shown in figure 8.7.

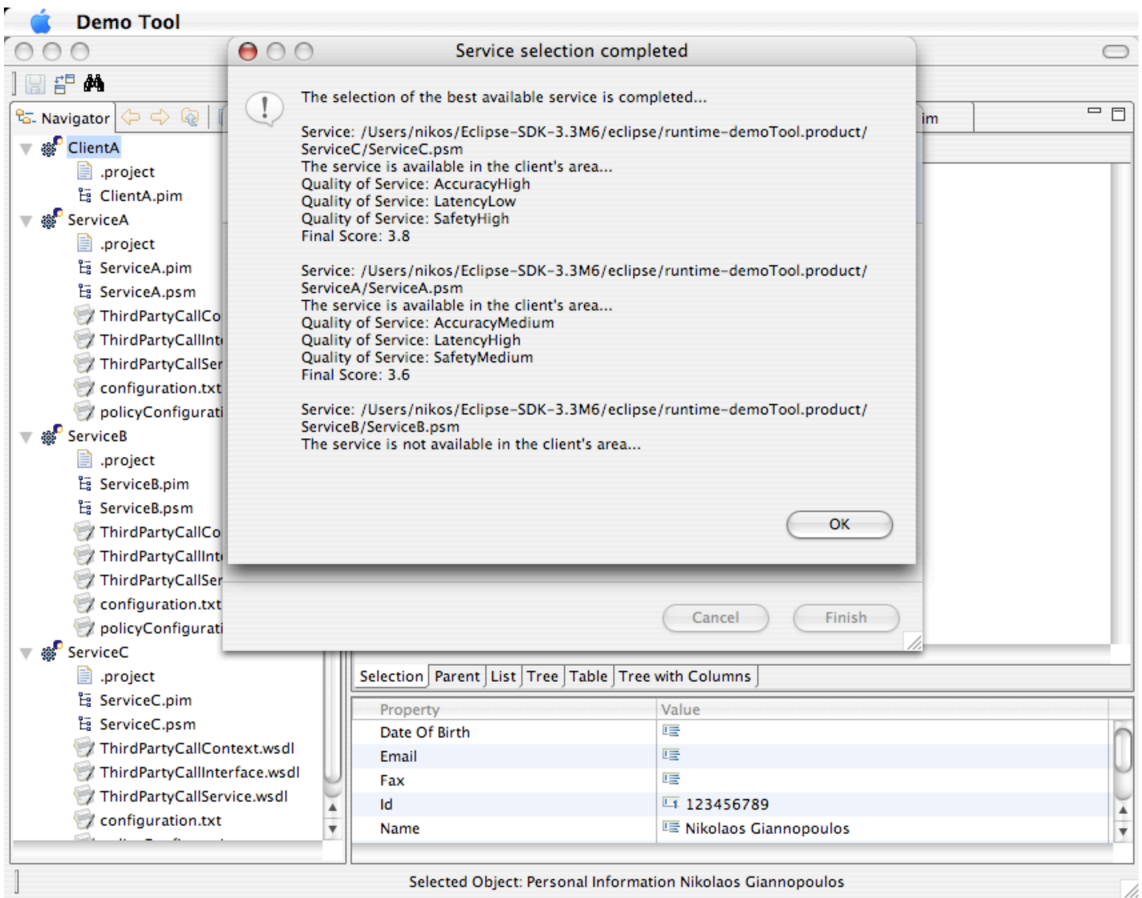


Figure 8.7: Service Selection Process Results

The best Web Service, according to the client's profile, is service A with score 3.6. The optimal configuration of the Web Service is Medium Accuracy, High Latency and Medium Safety. Service B was not available in the client's area, and service C had a worse score (3.8) compared to service A. However, the optimal configuration for service C is returned as well and it is High Accuracy, Low Latency and High Safety. This completes the demonstration of our framework.

Chapter 9

Conclusions and Future Work

The software engineering community has come to the realization that the current status of Web Service discovery and selection is inadequate for the contemporary needs of the businesses and their clients. The problem arises when conflicting Web Service descriptions, having almost identical syntactic descriptions, need to be evaluated when selecting the most appropriate Web Service according to the requirements posed by a specific client request. As of the time of this writing, there is no standard support for semantically annotating Web Service descriptions. The use of semantically enhanced Web Service descriptions could solve the problems arising when two or more services have similar descriptions, and furthermore, it would enable a more sophisticated querying of Web Services from their potential clients. The current trend in software development is the model-driven development, which will dominate in the years to come. As a result, a proposal addressing the issue of semantically annotating Web Services and selecting services using these annotations should be made in the context of model-driven development.

There are a number of proposals in the literature addressing similar issues, presented in Chapter 2; however, these proposals are mostly focused on specific areas of the problem domain without handling the problem as a whole. For example, there are proposals addressing the issue of model-driven development of semantically enriched Web Services using ontologies or extending WSDL, without, however, proposing a framework that would utilize these descriptions to select, in some way, the best Web Services according to the needs of the clients. There are other proposals addressing the issue of discovering Web Services in an environment in which Web Services are semantically enriched, without, however, proposing a framework to generate those semantically enriched descriptions.

We propose a concrete solution addressing the problem as a whole, and not just focusing on specific areas of the problem domain. In this way, an authority interested in our approach would have a full solution of the problem domain, without having to adopt a solution from one provider addressing the model-driven development of semantically enriched Web Service descriptions, and another solution addressing the discovery of these services from another provider. When combining solutions from different sources, there is a high risk of having interoperability and incompatibility issues. Our proposal, a model-driven service discovery framework for carrier applications, does not only provide a full proposal but is additionally fully extensible to accommodate future expansions as needed.

9.1 Thesis Contributions

The contributions of the thesis can be divided into two categories: the first category addresses the issue of generating semantically-enriched Web Service descriptions in a model-driven manner, and the second category addresses the issue of selecting semantically-enriched Web Services according to the preferences of the clients requesting the Web

Services.

In our effort to address the model-driven generation of semantically enriched Web Service descriptions, we specified a syntactic-based service interface description specification, a semantic-based service description specification and integrated these two specifications into a Platform Independent Model for service-oriented systems. In addition, we specified a Platform Specific Model for Web Services accommodating the specifications defined in the PIM in a platform-specific manner (WSDL-specific), extending the WSDL-specific definitions to include the semantic-specific definitions that are not included in the WSDL 1.1 specification. Furthermore, we designed a transformation model describing the transformation rules and mappings between the PIM model and the PSM model and created a concrete implementation of the transformation model using the ATL language. In order to generate the extended WSDL documents we designed and implemented a second transformer, transforming the generated PSM models into WSDL 1.1 code, containing all the syntactic and semantic information specified for the service.

In order to address the efficient selection of Web Services according to the specific needs of each client, we designed a PIM model defining a client's profile, including the client's location, his/her personal information and preferences. The created client profiles will be used when selecting the appropriate Web Services for a client, based on the semantic information included in the client's profile. We propose an adaptive service selection framework. The adaptiveness is justified with the introduction of the semantic information in both the service and the client descriptions, and additionally with the introduction of different policies defining the service selection process. We propose a framework for defining service selection policies, providing three predefined policies for demonstration purposes. The proposed framework allows the dynamic configuration of the Web Services to meet specific quality requirements defined by the clients.

The framework was specified in the context of Model Driven Development, targeting carrier applications, thus being an innovating approach for Web Service description, discovery and selection. For proof of concept and for demonstration purposes we designed and developed an Eclipse Rich Client Platform (RCP) prototype tool.

9.2 Future Work

The purpose of the proposed framework is to provide the foundation of an innovating approach to generate semantically enriched Web Service descriptions and at the same time select the most appropriate Web Services according to the needs of their clients. We designed the framework in such a way that can be easily extended to accommodate new ideas and new approaches in future releases. Both PIM and PSM metamodels can be extended if it is decided that the currently defined specifications do not completely cover the solution domain. The semantic specifications may be easily extended to include additional information. For example, someone may define additional QoS characteristics or use a different approach in specifying the locations of Web Services and clients by using coordinates instead of postal codes. New and more sophisticated policies and service selection algorithms may be specified, enabling an even more accurate and robust service selection methodology. Another interesting future extension would be to integrate our framework in workflows, facilitating the instantiation of templated business workflows, as presented in chapter 7.

Appendix A

Full Version of PIM

A.1 PIM Full UML Diagrams

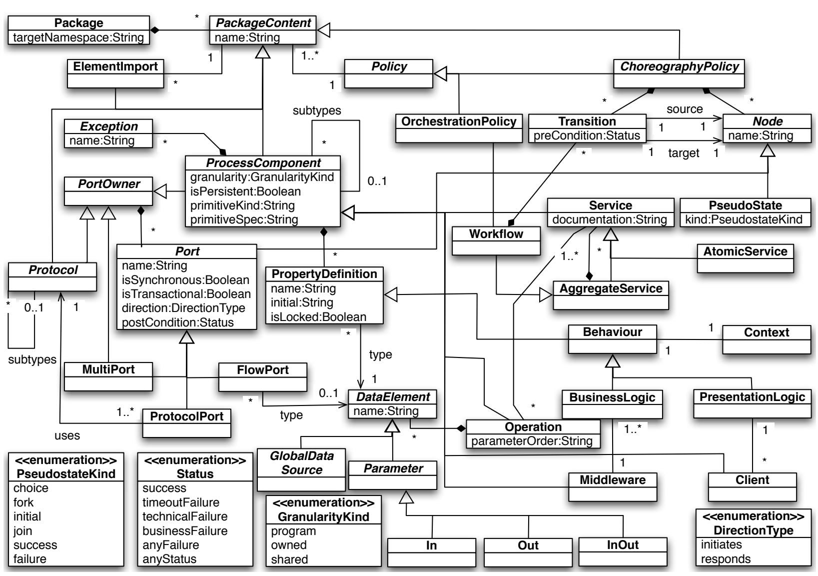


Figure A.1: Syntactic PIM Full Version Part 1

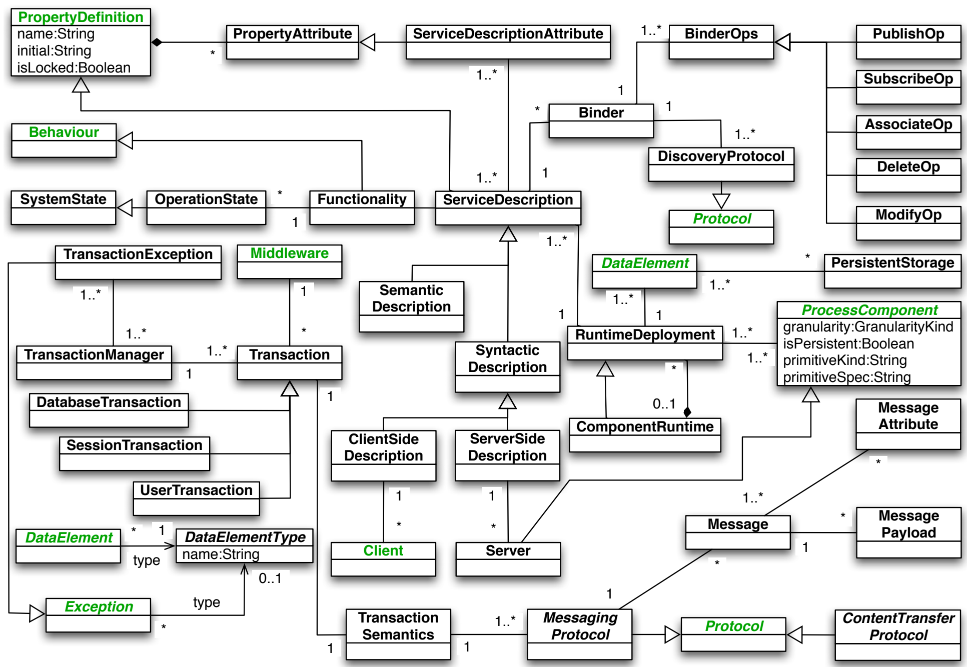


Figure A.2: Syntactic PIM Full Version Part 2

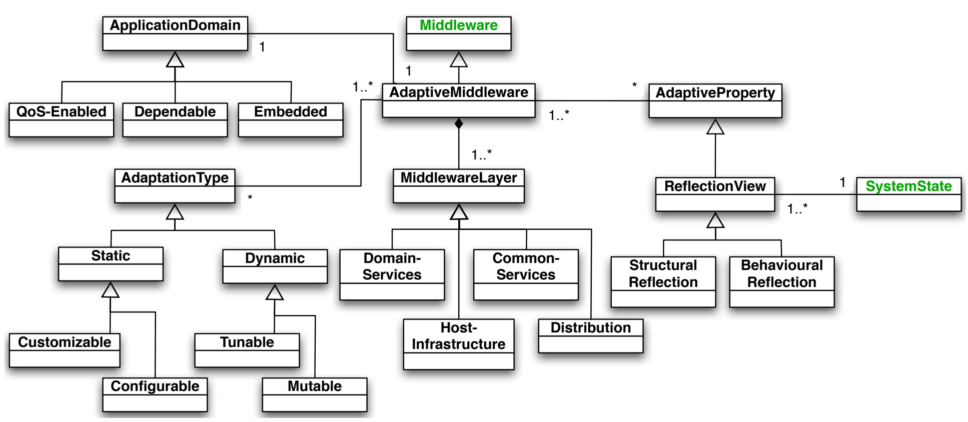


Figure A.3: Syntactic PIM Full Version Part 3

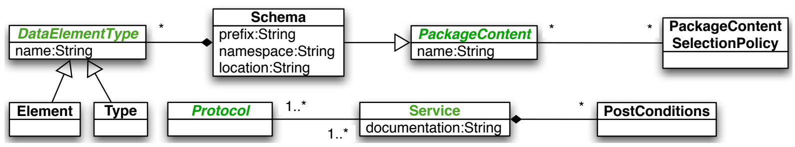


Figure A.4: Syntactic PIM Full Version Part 4

A.2 PIM Full Documentation

Table A.1: Syntactic PIM Full Version Documentation

Class Name	**Package**.
Semantics	Defines a structural container for "top level" model elements.
Extends	None.
Attributes	***targetNamespace: String (required)*** The namespace of the service definition.
Associations	***PackageContent (zero or more)*** The model element(s) within the package.

Class Name	**PackageContent (abstract)**.
Semantics	An abstract capability that represents an element that may be placed in a package and thus referenced from other elements of the package.
Extends	None.
Attributes	***name: String (optional)*** The name of the element.
Associations	***Policy (exactly one)*** The policy associated with each element. ***ElementImport (zero or more)*** The element(s) that might be imported into another package. ***PackageContentSelectionPolicy (zero or more)*** The selection policies associated with the element.

Continued on next page

Table A.1 – continued from previous page

Class Name	**ProcessComponent (abstract)**.
Semantics	A ProcessComponent represents an abstract active processing unit (it does something). Each ProcessComponent defines a set of ports for interaction with other ProcessComponents and has a set of properties that are used to configure the ProcessComponent when it is used.
Extends	***PackageContent***, ***PortOwner***.
Attributes	***granularity: GranularityKind (optional)*** The GranularityKind defines the scope in which the component operates. Its values may be: program (the component is local to a program instance (default)), owned (the component is visible outside of the scope of a particular program but dedicated to a particular task or session that controls its life cycle) and shared (the component is generally visible to external entities via some kind of distributed infrastructure). ***isPersistent: Boolean (optional)*** Indicates that the component stores session specific state across interactions. ***primitiveKind: String (optional)*** Components implementation includes additional implementation semantics defined elsewhere, perhaps in an action language or programming language. If the component has an implementation specification, primitiveKind specifies the implementation specific type, normally the name of a programming language. If primitiveKind is blank, the composition is the full specification of the components implementation (the component is not primitive). ***primitiveSpec: String (optional)*** If primitiveKind has a value, primitiveSpec identifies the location of the implementation. The syntax of primitiveKind is implementation specific.
Associations	***Port (zero or more) (via PortOwner)*** The set of Ports on the ProcessComponent. Each port provides a connection point for interaction with other components or services and realizes a specific protocol. The protocol may be simple and use a "FlowPort" or the protocol may be complex and use a "ProtocolPort". If allowed by its protocol, a port may send and receive information.
	Continued on next page

Table A.1 – continued from previous page

	Supertype (zero or one), Subtypes (zero or more) A ProcessComponent may inherit specification elements (ports, properties, states etc.) from a supertype. That supertype must also be a ProcessComponent. A subtype component is bound by the contract of its supertypes but it may add elements, override property values, and restrict referenced types. ***PropertyDefinition (zero or more)*** To make a component capable of being reused in a variety of conditions it is necessary to be able to define and set properties of that component. PropertyDefinition represents the list of properties defined for this component. ***Exception (zero or more)*** The exception(s) associated with the ProcessComponent. ***RuntimeDeployment (one or more)*** The runtime deployment environment(s) associated with the ProcessComponent.
Class Name	**Exception (abstract)**.
Semantics	When defining a service it is useful to declare the exceptions that may be thrown or events that may occur as a result of an erroneous state.
Extends	None.
Attributes	***name: String (required)*** The name of the exception.
Associations	***DataElementType (zero or one)*** When the exception is a data type then this association is used to define its type (*e.g.*, XML schema element, simpleType, complexType, *etc.*).
Class Name	**TransactionException**.
Semantics	The exceptions thrown during the execution of transactions.
Extends	***Exception***.
Attributes	None.
Associations	***TransactionManager (one or more)*** The transaction manager(s) handling the exception.

Continued on next page

Table A.1 – continued from previous page

Class Name	**Schema**.
Semantics	Represents an external imported XML schema declaration to be used inside a WSDL document. The elements of a schema are used when defining a service, for example when defining the parameters of the operations.
Extends	***PackageContent***.
Attributes	***prefix: String (required)*** The prefix used when referencing the schema. ***namespace: String (required)*** The URI representing the "targetNamespace" attribute of a schema. ***location: String (optional)*** When importing a schema as an external document the "location" attribute is used to indicate the location, in the local file system, of the file containing the schema. If it is omitted it means that the schema is accessible over the Internet.
Associations	***DataElementType (zero or more)*** The XML schema "element" or "type" element(s) declared inside the schema. These elements will be used when defining the service.
Class Name	**Element**.
Semantics	Represents the "element" element in an XML schema document. The "complexType" and "simpleType" elements define data types, not actual data elements. The distinction between these two is analogous to the difference between a class and an instance of that class. The data elements are defined using the "element" element.
Extends	***DataElementType***.
Attributes	None.
Associations	None.
Class Name	**Type**.
Semantics	The complexType and simpleType elements inside an XML schema declaration define data types. The class is an abstract grouping of both complex and simple types.
Extends	***DataElementType***.
Attributes	None.
Associations	None.

Continued on next page

Table A.1 – continued from previous page

Class Name	**Service**.
Semantics	In a service-oriented architecture, we need a clear understanding of the term service. This is achieved by defining the class Service.
Extends	***ProcessComponent***.
Attributes	***documentation: String (optional)*** A brief documentation (description) of the service.
Associations	***Operation (zero or more)*** The operation(s) of the service. ***PostConditions (zero or more)*** The post-conditions of the service. The post-conditions describe the effects of the service after it is executed. ***Protocol (one or more)*** The messaging, discovery or content transfer protocols associated with the service. ***PreConditions (zero or one)*** The preconditions of the service that need to be satisfied before the service is executed. This is part of the semantic information of the service and as a result the association is shown in the semantic metamodel presented in Chapter 4, in figure 4.1. ***ServiceContext (zero or one)*** The context of the service. This is part of the semantic information of the service and as a result the association is shown in the semantic metamodel presented in Chapter 4, in figure 4.1. ***PointsOfAvailability (zero or more)*** The location(s) the service is available. This is part of the semantic information of the service and as a result the association is shown in the semantic metamodel presented in Chapter 4, in figure 4.2. ***ServiceLocation (zero or one)*** The physical location of the service. This is part of the semantic information of the service and as a result the association is shown in the semantic metamodel presented in Chapter 4, in figure 4.2.
Class Name	**Operation**.
Semantics	Represents an operation of a service.
Extends	***ProcessComponent***.

Continued on next page

Table A.1 – continued from previous page

Attributes	***parameterOrder: String (optional)*** Operations do not specify whether they are to be used with RPC-like bindings or not. However, when using an operation with an RPC-binding, it is useful to be able to capture the original RPC function signature. For this reason, an operation may specify an order of parameter names via the "parameterOrder" attribute. The value of the attribute is a list of message part names separated by a single space. Note that this information serves as a "hint" and may safely be ignored by those not concerned with RPC signatures. Also, it is not required to be present, even if the operation is to be used with an RPC-like binding.
Associations	***Service (one or more)*** The service(s) containing the operation. ***DataElement (zero or more)*** The data element(s) associated with the operation.
Class Name	**DataElement (abstract)**.
Semantics	DataElement is the abstract super type of all parameters and global data sources defined and used in the service. It defines some kind of information.
Extends	None.
Attributes	***name: String (required)*** The name of the data element.
Associations	***DataElementType (exactly one)*** The type of the data element. ***PersistentStorage (zero or more)*** The persistent storage medium(s) (e.g. databases, files) in which the DataElement is stored. ***RuntimeDeployment (exactly one)*** The runtime deployment environment in which the data element is used.
Class Name	**DataElementType (abstract)**.
Semantics	DataElementType is the abstract super type of all elements that can be types of a DataElement.
Extends	None.

Continued on next page

Table A.1 – continued from previous page

Attributes	***name: String (required)*** The name of the type.
Associations	None.
Class Name	**Parameter (abstract)**.
Semantics	The abstract super type of the parameters of an operation.
Extends	***DataElement***.
Attributes	None.
Associations	None.
Class Name	**GlobalDataSource (abstract)**.
Semantics	An abstract class representing data sources that have a more global character than the parameters of an operation.
Extends	***DataElement***.
Attributes	None.
Associations	None.
Class Name	**InOut**.
Semantics	Represents the "in/out" parameters. If a parameter appears in both the input and output, it is an "in/out" parameter. The value of an "in/out" argument is sent in the input and is modified from the reply. An "in/out" argument is therefore both an "in" and "out" argument.
Extends	***Parameter***.
Attributes	None.
Associations	None.
Class Name	**In**.
Semantics	Represents the "in" parameters. If a parameter appears in only the input, it is an "in" parameter. "In" arguments are sent in the input but do not change as a result of the method invocation. This is a direct mapping of the pass-by-value semantics of arguments in Java method calls.
Extends	***Parameter***.
Attributes	None.
Associations	None.

Continued on next page

Table A.1 – continued from previous page

Class Name	**Out**.
Semantics	Represents the "out" parameters. If a parameter appears in only the output message, it is an "out" parameter. "Out" arguments appear in the method signature, but their value is not sent with the input message. However, a new value for the argument may appear in the response and, if so, the argument is modified from the returned value.
Extends	***Parameter***.
Attributes	None.
Associations	None.
Class Name	**Client**.
Semantics	Represents the possible clients that may query for or use a service.
Extends	***ProcessComponent***.
Attributes	None.
Associations	***PresentationLogic (exactly one)*** The PresentationLogic associated with the specific client. ***ClientSideDescription (exactly one)*** The client's syntactic description of the service. ***ClientProfile (zero or one)*** The profile of the client. This is part of the semantic information of the client and as a result the association is shown in the semantic metamodel presented in Chapter 4, in figure 4.1.
Class Name	**Port (abstract)**.
Semantics	A port realizes a simple or complex conversation for a ProcessComponent or Protocol. All interactions with a ProcessComponent are done via one of its ports. When a component is instantiated, each of its ports is instantiated as well, providing a well-defined connection point for other components. Each port is connected with collaborative components that speak the same protocol. Multi-party conversions are defined by components using multiple ports, one for each kind of party.
Extends	***Node***.
Attributes	***name: String (required)*** The name of the port. The name will, by default, be the same as the name of the protocol role it realizes.

Continued on next page

Table A.1 – continued from previous page

	isSynchronous: Boolean (required) A port may interact synchronously or asynchronously. A port, which is marked as synchronous, is required to interact using synchronous messages and return values. ***isTransactional: Boolean (required)*** Indicates that interactions with the component are transactional and atomic (in most implementations this will require that a transaction be started on receipt of a message). Non-transactional components either maintain no state or must execute within a transactional component. ***direction: DirectionType (required)*** Indicates that the port will either initiate or respond to the related type. An initiating port will send the first message. Note that by using ProtocolPorts a port may be the initiator of some protocols and the responder to others. The values of DirectionType may be initiates (this port will initiate the conversation by sending the first message), or responds (this port will respond to the initial message and (potentially) continue the conversation). ***postCondition: Status (optional)*** The status of the conversation indicated by the use of this port. This status may be queried in the postCondition of a transition.
Associations	None.
Class Name	**FlowPort**.
Semantics	A FlowPort is a port that defines a data flow in or out of the port on behalf of the owning component or protocol.
Extends	***Port***.
Attributes	None.
Associations	***DataElement (zero or one)*** The type of data element that may flow into or out of the port.
Class Name	**ProtocolPort**.
Semantics	A protocol port is a port that defines the use of a protocol. A protocol port is used for potentially complex two-way interactions between components, such as is common in B2B protocols. Since a protocol has two "roles" (the initiator and responder), the direction is used to determine which role the protocol port is taking on.
	Continued on next page

Table A.1 – continued from previous page

Extends	***Port***.
Attributes	None.
Associations	***Protocol (exactly one)*** The protocol to use, which becomes the specification of this port's behavior.

Class Name	**MultiPort**.
Semantics	A MultiPort combines a set of ports which are behaviourally related. Each port owned by the MultiPort will "buffer" information sent to that port until all the ports within the MultiPort have received data, at this time all the ports will send their data. Owned ports will not forward data until all sub-ports have received data.
Extends	***Port***, ***PortOwner***.
Attributes	None.
Associations	None.

Class Name	**Protocol (abstract)**.
Semantics	A protocol defines a type of conversation between two parties, the initiator and responder. One protocol role is the initiator of the conversation and the other the responder. However, after the conversation has been initiated, individual messages and sub-protocols may be initiated by either party. The ports of a protocol are specified with respect to the responder. Within the protocol are sub-ports. Each port contained by a protocol defines a sub-action of that protocol until ultimately everything is defined in terms of FlowPorts. A protocol must be used by two ProtocolPorts to become active. The protocol specifies the conversation between two ProcessComponents (via their ports). Each component that is using that protocol must use it from the perspective of the "initiating role" or the "responding role". Each of these components will use every port in the protocol, but in complementary directions. For example, a protocol "X" has a flow port "A" that initiates a message and a flow port "B" that responds to a message. Component "Y", which responds to protocol "X" will also receive "A" and initiate "B". But, Component "Z", which initiates protocol "X" will initiate message "A" and respond to message "B".
Extends	***PackageContent***, ***PortOwner***.
	Continued on next page

Table A.1 – continued from previous page

Attributes	None.
Associations	***Port (zero or more) (via PortOwner)*** The ports, which define the sub-actions of the protocol. For example, a "call Return" protocol may have a "call" FlowPort and a "return" FlowPort. ***Supertype (zero or one), Subtypes (zero or more)*** A Protocol may inherit specification elements from a supertype. That supertype must also be a Protocol. ***Service (one or more)*** The service(s) using the protocol.
Class Name	**PropertyDefinition**.
Semantics	To allow for greater flexibility and reuse, ProcessComponents may have properties that may be set when the ProcessComponent is used. A PropertyDefinition defines that such a property exists, its name, and type.
Extends	None.
Attributes	***name: String (required)*** The name of the property being modeled. ***initial: String (optional)*** An expression indicating the initial and default value. ***isLocked: Boolean (optional)*** If true, the property may not be changed.
Associations	***DataElement (exactly one)*** The type of the property. ***PropertyAttribute (zero or more)*** The PropertyAttribute(s) owned by the PropertyDefinition.
Class Name	**PortOwner (abstract)**.
Semantics	An abstract meta-class used to group the meta-classes that may own ports: ProcessComponent, Protocol and MultiPort.
Extends	None.
Attributes	None.
Associations	***Port (zero or more)*** The owned ports.

Continued on next page

Table A.1 – continued from previous page

Class Name	**ElementImport**.
Semantics	Defines an "alias" for one element within another package.
Extends	***PackageContent***.
Attributes	None.
Associations	***PackageContent (exactly one)*** The element to be imported.
Class Name	**Policy (abstract)**.
Semantics	Describes the notion of policies that may be specified.
Extends	None.
Attributes	None.
Associations	***PackageContent (one or more)*** The elements belonging to the policy.
Class Name	**ChoreographyPolicy (abstract)**.
Semantics	An abstract class that owns Nodes and Transitions. A Choreography-Policy specifies the ordering of port activities. The order in which actions of the ProcessComponent's ports do something may be specified using ChoreographyPolicy. The ChoreographyPolicy of a ProcessComponent specifies the external temporal contract of the ProcessComponent (when it will do what) based on the actions of its ports and the ports in protocols of its ports.
Extends	***PackageContent***, ***Policy***.
Attributes	None.
Associations	***Node (zero or more)*** The Port(s) and/or PseudoState(s) to be choreographed. ***Transition (zero or more)*** The transition(s) between nodes.
Class Name	**Node (abstract)**.
Semantics	Node is an abstract element that specifies something that can be the source and/or target of a transition and thus ordered within the choreographed process.
Extends	None.
Attributes	***name: String (required)*** The name of the node.
Associations	None.

Continued on next page

Table A.1 – continued from previous page

Class Name	**Transition**.
Semantics	The contractual specification that the related nodes will activate based on the ordering imposed by the set of transitions between nodes.
Extends	None.
Attributes	***preCondition: Status (required)*** A constraint on the transition such that it may only fire if the prior node terminated with the referenced condition.
Associations	***Source Node (exactly one)*** The source node that is transferring control and/or data. ***Target Node (exactly one)*** The target node to which data and/or control will be transferred.

Class Name	**PseudoState**.
Semantics	PseudoState specifies starting, ending, or intermediate states in the ChoreographyPolicy of the contract of a protocol or ProcessComponent.
Extends	***Node***.

Continued on next page

Table A.1 – continued from previous page

Attributes	***kind: PseudostateKind (required)*** • Choice: splits an incoming transition into several disjoint outgoing transitions. Each outgoing transition has a guard condition that is evaluated after prior actions on the incoming path have been completed. At least one outgoing transition must be enabled or the model is ill-formed. • Fork: splits an incoming transition into several concurrent outgoing transitions. All the transitions fire together. • Initial: the default target of a transition to the enclosing composite state. • Join: merges transitions from concurrent regions into a single outgoing transition. Join PseudoState will proceed after all its incoming Transitions have triggered. • Success: the end-state indicating that the choreography ended in success. • Failure: the end-state indicating that the choreography ended in failure.
Associations	None.
Class Name	**OrchestrationPolicy**.
Semantics	The orchestration policy associated with a workflow.
Extends	***Policy***.
Attributes	None.
Associations	***Workflow*** The workflow for which the policy is defined.
Class Name	**AtomicService**.
Semantics	An atomic service is a function that is well-defined, self-contained, and does not depend on the context or state of other services. It cannot be decomposed into sub-services.
Extends	***Service***.

Continued on next page

Table A.1 – continued from previous page

Attributes	None.
Associations	None.
Class Name	**AggregateService**.
Semantics	A composition of atomic services or other aggregate services in order to form and define a new service.
Extends	***Service***.
Attributes	None.
Associations	***Service (zero or more)*** The services contained in the aggregate service.
Class Name	**Workflow**.
Semantics	A workflow is a formal definition of the steps required by a process and the sequence in which the steps occur. In the context of our metamodel it defines the execution order of a set of services.
Extends	***AggregateService***.
Attributes	None.
Associations	***Transition (zero or more)*** The Transition(s) contained in the workflow. ***OrchestrationPolicy*** The OrchestrationPolicy defined for the workflow.
Class Name	**Behaviour**.
Semantics	Describes the behaviour of ProcessComponents (*e.g.*, services, middleware *etc.*) either from the middleware perspective (business logic) or from the perspective of a client (presentation logic).
Extends	***PropertyDefinition***.
Attributes	None.
Associations	***Context (exactly one)*** The context under which the behaviour exists.
Class Name	**Context**.
Semantics	The context under which the behaviour of a ProcessComponent exists (*e.g.*, location of client).
Extends	None.
Attributes	None.

Continued on next page

Table A.1 – continued from previous page

Associations	***Behaviour (exactly one)*** The behaviour that corresponds to the specific context.

Class Name	**BusinessLogic**.
Semantics	The behaviour of a middleware from the perspective of a middleware. BusinessLogic may include where the business logic is, what communications are taking place, what transactions are failing, how many are being handled, what peers it depends on are not behaving *etc*. All the critical knowledge about what is happening is in the business logic.
Extends	***Behaviour***.
Attributes	None.
Associations	***Middleware (exactly one)*** The middleware associated with the specific BusinessLogic.

Class Name	**Middleware**.
Semantics	Middleware is connectivity software that encapsulates a set of services residing above the network operating system layer and below the user application layer. Middleware facilitates the communication and coordination of application components that are potentially distributed across several networked hosts. Moreover, middleware provides application developers with high-level programming abstractions, for example, use of remote objects instead of socket programming. In this manner, middleware can hide interprocess communication, mask the heterogeneity of the underlying systems (hardware devices, operating systems, and network protocols), and facilitate the use of multiple programming languages at the application level. Middleware can also be considered as "glue" that enables integration of legacy applications, effectively implementing the session and presentation layers (layers 5 and 6) of the ISO OSI reference model.
Extends	***ProcessComponent***.
Attributes	None.
Associations	***BusinessLogic (one or more)*** The BusinessLogic(s) specified for the middleware. ***Transaction (zero or more)*** The transaction(s) associated with the middleware.

Continued on next page

Table A.1 – continued from previous page

Class Name	**PresentationLogic**.
Semantics	How to present the information (*e.g.*, VoiceXML).
Extends	***Behaviour***.
Attributes	None.
Associations	***Client (zero or more)*** The client(s) associated with the specific PresentationLogic.
Class Name	**PropertyAttribute**.
Semantics	A set of attributes associated with one or more PropertyDefinitions.
Extends	None.
Attributes	None.
Associations	None.
Class Name	**ServiceDescriptionAttribute**.
Semantics	A set of attributes associated with one or more ServiceDescriptions.
Extends	***PropertyAttribute***.
Attributes	None.
Associations	***ServiceDescription (one or more)*** The service description(s) associated with the attribute.
Class Name	**SystemState**.
Semantics	Describes a state of a system.
Extends	None.
Attributes	None.
Associations	***ReflectionView (one or more)*** The reflection view(s) associated with a given state.
Class Name	**OperationState**.
Semantics	A state of a system related to its operation.
Extends	***SystemState***.
Attributes	None.
Associations	***Functionality (exactly one)*** The functionality of a service that the OperationState is associated with.
Class Name	**Functionality**.
Semantics	The functionality of a service.
Extends	***Behaviour***.

Continued on next page

Table A.1 – continued from previous page

Attributes	None.
Associations	***OperationState (zero or more)*** The operational state(s) associated with the functionality. ***ServiceDescription (zero or more)*** The service description(s) associated with the specific functionality.
Class Name	**ServiceDescription**.
Semantics	The description of a service. The description can be either syntactic or semantic. For example, consider a stock quote service, which takes as input a string denoting the stock symbol and returns the stock quote as a number. The syntactic information denotes that the input parameter is a string and the output is a number, whereas semantic information conveys the real world meaning of the string and the number in the context of stock quote markets. Depending on whether the service requestor is an end-user, a developer or a machine, different kinds of service description are required. For the end-user, only semantic description is needed whereas developers or machines need both semantic and syntactic information.
Extends	***PropertyDefinition***.
Attributes	None.
Associations	***ServiceDescriptionAttribute (one or more)*** The attribute(s) of the service description. ***Functionality (zero or more)*** The functionality(s) associated with the service. ***Binder (zero or more)*** The Binder(s) attached to a specific ServiceDescription. ***RuntimeDeployment (exactly one)*** The runtime deployment environment associated with the specific ServiceDescription.
Class Name	**Binder**.
Semantics	Binds a service description to one or more discovery protocols.
Extends	None.
Attributes	None.
	Continued on next page

Table A.1 – continued from previous page

Associations	***ServiceDescription (exactly one)*** The service description on which the Binder is attached. ***DiscoveryProtocol (one or more)*** The discovery protocol(s) on which the Binder is attached. ***BinderOps (one or more)*** The set of operations associated with the Binder.
Class Name	**DiscoveryProtocol**.
Semantics	Used in UDDI (Universal Description, Discovery and Integration). The protocol used to discover web services.
Extends	***Protocol***.
Attributes	None.
Associations	***Binder (exactly one)*** The Binder attached to the discovery protocol.
Class Name	**BinderOps**.
Semantics	A set of operations associated with a binder.
Extends	None.
Attributes	None.
Associations	***Binder (exactly one)*** The Binder associated with the operations.
Class Name	**PublishOp**.
Semantics	The operation of publishing a service.
Extends	***BinderOps***.
Attributes	None.
Associations	None.
Class Name	**SubscribeOp**.
Semantics	The operation of subscribing for a service.
Extends	***BinderOps***.
Attributes	None.
Associations	None.
Class Name	**AssociateOp**.
Semantics	The operation of associating with a service.
Extends	***BinderOps***.
Attributes	None.

Continued on next page

Table A.1 – continued from previous page

Associations	None.
Class Name	**DeleteOp**.
Semantics	The operation of deleting a service.
Extends	***BinderOps***.
Attributes	None.
Associations	None.
Class Name	**ModifyOp**.
Semantics	The operation of modifying a service.
Extends	***BinderOps***.
Attributes	None.
Associations	None.
Class Name	**PersistentStorage**.
Semantics	The persistent storage mediums (*e.g.*, databases, files) used to store information (DataElement(s)).
Extends	None.
Attributes	None.
Associations	***DataElement (one or more)*** The data element(s) stored in the persistent storage medium.
Class Name	**RuntimeDeployment**.
Semantics	The runtime deployment environment in which the middleware, the web services, the databases and the clients interact.
Extends	None.
Attributes	None.
Associations	***DataElement (one or more)*** The exchanged data element(s) that participate in the operation of the runtime deployment environment. ***ServiceDescription (one or more)*** The service description(s) associated with the runtime deployment environment. ***ProcessComponent (one or more)*** The ProcessComponent(s) associated with the runtime deployment environment.

Continued on next page

Table A.1 – continued from previous page

Class Name	**ComponentRuntime**.
Semantics	ComponentRuntime is a collection of runtime deployment environments.
Extends	***RuntimeDeployment***.
Attributes	None.
Associations	***RuntimeDeployment (zero or more)*** The runtime deployment environments contained in the ComponentRuntime.
Class Name	**SyntacticDescription**.
Semantics	Syntactic information is concerned with the implementation aspects of a service and thus tailored towards the programmers' requirements. Associated with WSDL.
Extends	***ServiceDescription***.
Attributes	None.
Associations	None.
Class Name	**SemanticDescription**.
Semantics	Semantic information is concerned with the conceptual aspects of a service aiming to facilitate end-users by shielding off the lower level technical details, as well as to facilitate developers to find services that best match their needs and to enable automatic service selection and composition.
Extends	***ServiceDescription***.
Attributes	None.
Associations	None.
Class Name	**ClientSideDescription**.
Semantics	The SyntacticDescription from the perspective of a client.
Extends	***SyntacticDescription***.
Attributes	None.
Associations	***Client (zero or more)*** The client(s) associated with this description.
Class Name	**ServerSideDescription**.
Semantics	The SyntacticDescription from the perspective of a server. Related with the web-service deployment.

Continued on next page

Table A.1 – continued from previous page

Extends	***SyntacticDescription***.
Attributes	None.
Associations	***Server (zero or more)*** The server(s) associated with this description.

Class Name	**Server**.
Semantics	The server in a service-oriented environment.
Extends	***ProcessComponent***.
Attributes	None.
Associations	***ServerSideDescription (exactly one)*** The server's syntactic description of the service.

Class Name	**TransactionManager**.
Semantics	The manager responsible for handling transactions.
Extends	None.
Attributes	None.
Associations	***TransactionException (one or more)*** The transactional exception(s) possibly thrown by the manager. ***Transaction (one or more)*** The transactions handled by the manager.

Class Name	**Transaction**.
Semantics	A transaction occurring in a service-oriented environment.
Extends	None.
Attributes	None.
Associations	***TransactionManager (exactly one)*** The transaction manager handling the transaction. ***Middleware (exactly one)*** The middleware associated with the transaction. ***TransactionSemantics (exactly one)*** The semantic description of the transaction.

Class Name	**DatabaseTransaction**.
Semantics	The transactions performed in the context of databases. Mostly used in conjunction with Java Database Connectivity (JDBC). A JDBC transaction is controlled by the transaction manager of the DBMS.
Extends	***Transaction***.

Continued on next page

Table A.1 – continued from previous page

Attributes	None.
Associations	None.
Class Name	**SessionTransaction**.
Semantics	Related to Bean-Managed Transactions. In a bean-managed transaction, the code in the session or message-driven bean explicitly marks the boundaries of the transaction.
Extends	***Transaction***.
Attributes	None.
Associations	None.
Class Name	**UserTransaction**.
Semantics	The transactions performed using HTTP requests by the users.
Extends	***Transaction***.
Attributes	None.
Associations	None.
Class Name	**TransactionSemantics**.
Semantics	The semantic description of a transaction.
Extends	None.
Attributes	None.
Associations	***Transaction (exactly one)*** The transaction described by the semantics. ***MessagingProtocol (one or more)*** The messaging protocol(s) associated with the semantic description of a transaction.
Class Name	**MessagingProtocol (abstract)**.
Semantics	The messaging protocol used in a transaction. Usually it is a SOAP protocol.
Extends	***Protocol***.
Attributes	None.
Associations	***TransactionSemantics (exactly one)*** The transactional semantics associated with the protocol. ***Message (zero or more)*** The message(s) exchanged in the context of a protocol.

Continued on next page

Table A.1 – continued from previous page

Class Name	**Message**.
Semantics	The message(s) exchanged in a service-oriented environment.
Extends	None.
Attributes	None.
Associations	***MessagingProtocol (exactly one)*** The protocol defining the messages role. ***MessageAttribute (zero or more)*** The attribute(s) of the message. ***MessagePayload (zero or more)*** The information contained in the message.
Class Name	**MessageAttribute**.
Semantics	The attributes of a message.
Extends	None.
Attributes	None.
Associations	***Message (one or more)*** The message(s) containing the attribute.
Class Name	**MessagePayload**.
Semantics	The information contained in a message.
Extends	None.
Attributes	None.
Associations	***Message (exactly one)*** The message containing the information.

Continued on next page

Table A.1 – continued from previous page

Class Name	**AdaptiveMiddleware**.
Semantics	Developing distributed applications is a difficult task for several reasons. First, writing code for interprocess communications is tedious and error prone. Low level socket programming and marshalling and un-marshalling messages are examples of such code. Second, supporting multiple interacting platforms is difficult. Many heterogeneous hardware devices, computer networks, operating systems, and programming languages have emerged during the last two decades. Distributed applications are more likely than stand-alone applications to involve heterogeneous technologies. Third, adapting to dynamic changing conditions is hard to achieve without the right tools and techniques. Emerging distributed applications often involve multimedia communication, mobility, embedded computing, group communications, and high availability. Addressing these issues means that systems must adapt to changing conditions, such as unexpected security attacks, hardware failures, and dynamic network environments. To tackle the first two problems, middleware was invented. Traditionally, middleware hides the underlying details of interprocess communication and heterogeneous technologies from the application developers using a "black-box" paradigm such as encapsulation in object-oriented programming. Although traditional middleware solves these problems to some extent, it is limited in its ability to support adaptation. Adaptive middleware has evolved from traditional middleware to solve all the three problems together. Adaptive middleware enables modifying the behaviour of a distributed application after the application is developed in response to some changes in functional requirements or operating conditions.
Extends	***Middleware***.
Attributes	None.
Associations	***ApplicationDomain (exactly one)*** The application domain of the middleware. ***AdaptiveProperty (zero or more)*** The adaptive properties of the middleware. ***AdaptionType (zero or more)*** The adaptation type(s) characterizing the middleware.

Continued on next page

Table A.1 – continued from previous page

	MiddlewareLayer (one or more) The layers composing the middleware.
Class Name	**AdaptiveProperty**.
Semantics	In addition to the foundation provided by the design and use of traditional middleware platforms, numerous advances in programming paradigms have also contributed to the design of adaptive middleware. The class represents the properties that play key roles in supporting adaptive middleware.
Extends	None.
Attributes	None.
Associations	***AdaptiveMiddleware (one or more)*** The adaptive middleware having the property.
Class Name	**ReflectionView**.
Semantics	Reflection refers to the ability of a program to reason about, and possibly alter its own behaviour or structure. Reflection enables a system to "open up" its implementation details for such analysis without compromising portability or revealing the unnecessary parts. In other words, reflection exposes a system implementation at a level of abstraction that hides unnecessary details, but still enables changes to the system behaviour or structure.
Extends	***AdaptiveProperty***.
Attributes	None.
Associations	***SystemState (exactly one)*** The systems state associated with the reflection property.
Class Name	**StructuralReflection**.
Semantics	Structural reflection enables modifying the structure of a system.
Extends	***ReflectionView***.
Attributes	None.
Associations	None.
Class Name	**BehaviouralReflection**.
Semantics	Behavioural reflection enables modifying the behaviour of a system (*e.g.*, encrypting requests before transmitting them over a network).
Extends	***ReflectionView***.
	Continued on next page

Table A.1 – continued from previous page

Attributes	None.
Associations	None.
Class Name	**MiddlewareLayer**.
Semantics	The layers a middleware is composed of.
Extends	None.
Attributes	None.
Associations	None.
Class Name	**Host-Infrastructure**.
Semantics	The host-infrastructure layer resides directly atop the operating system kernel and provides a higher-level API than the operating system API that hides the heterogeneity of hardware platforms, operating systems and, to some extent, network protocols. Host-infrastructure middleware provides generic services to the upper middleware layers by encapsulating functionality that would otherwise require much tedious, error-prone, and non-portable code, such as socket programming and thread communication primitives.
Extends	***MiddlewareLayer***.
Attributes	None.
Associations	None.
Class Name	**Distribution**.
Semantics	The distribution layer resides atop the host-infrastructure layer and provides a high-level programming abstraction, such as remote method invocation, to application developers. Using the distribution layer, developers can write distributed applications similar to stand-alone applications. Moreover, this layer hides the heterogeneity of network protocols and, to some extent, the heterogeneity of operating systems and programming languages.
Extends	***MiddlewareLayer***.
Attributes	None.
Associations	None.

Continued on next page

Table A.1 – continued from previous page

Class Name	**Common-Services**.
Semantics	The common-services layer resides atop the distribution layer and provides functionality such as fault tolerance, security, load balancing, event propagation, logging, persistence, real-time scheduling, and transactions. The high-level services provided in this layer can be reused in different applications.
Extends	***MiddlewareLayer***.
Attributes	None.
Associations	None.
Class Name	**Domain-Services**.
Semantics	The domain-services layer resides atop the common-services layer and is tailored to a specific class of distributed applications. Unlike the common-services layer, the high-level services in this layer can be reused only for a specific domain.
Extends	***MiddlewareLayer***.
Attributes	None.
Associations	None.
Class Name	**AdaptionType**.
Semantics	Adaptive middleware can be categorized with respect to the type of adaptation it provides. If middleware enables adaptation during the application compile or startup time, we call it static middleware. If middleware enables adaptation during the application run time, we call it dynamic middleware. Static middleware is divided further into customizable and configurable middleware. Dynamic middleware can be divided into tunable and mutable middleware.
Extends	None.
Attributes	None.
Associations	***AdaptiveMiddleware (one or more)*** The adaptive middleware associated with the specific type.
Class Name	**Static**.
Semantics	If middleware enables adaptation during the application compile or startup time, we call it static middleware.
Extends	***AdaptionType***.
Attributes	None.

Continued on next page

Table A.1 – continued from previous page

Associations	None.
Class Name	**Dynamic**.
Semantics	If middleware enables adaptation during the application run time, we call it dynamic middleware.
Extends	***AdaptionType***.
Attributes	None.
Associations	None.
Class Name	**Customizable**.
Semantics	Customizable middleware enables adapting an application during the application compile (or link) time so that a developer can generate customized (adapted) versions of the application. Note that a customized version is generated in response to the functional and environmental changes realized after the application development time.
Extends	***Static***.
Attributes	None.
Associations	None.
Class Name	**Configurable**.
Semantics	Configurable middleware enables adapting an application during the application startup time, enabling an administrator to configure the middleware in response to the functional and environmental changes realized after the application compile time.
Extends	***Static***.
Attributes	None.
Associations	None.
Class Name	**Tunable**.
Semantics	Tunable middleware enables adapting an application after the application startup time (but before the application is actually being used). Doing so enables an administrator to fine-tune the application in response to the functional and environmental changes that occur after the application is started. We also define a variation of tunable middleware, repeatedly-tunable middleware that enables repeated-tuning of applications during run time.
Extends	***Dynamic***.

Continued on next page

Table A.1 – continued from previous page

Attributes	None.
Associations	None.

Class Name	**Mutable**.
Semantics	Mutable middleware is the most powerful type of adaptive middleware that enables adapting an application during run time. Hence, the middleware can be dynamically adapted while it is being used. The main difference between tunable middleware and mutable middleware is that in the former, the middleware core remains intact during the tuning process whereas in the latter, there is no concept of fixed middleware core. Therefore, mutable middleware are more likely to evolve to something completely different and unexpected.
Extends	***Dynamic***.
Attributes	None.
Associations	None.

Class Name	**ApplicationDomain**.
Semantics	Categorizes adaptive middleware with respect to application domain. Most adaptive middleware projects support one of these three main application domains: QoS-oriented systems, dependable systems, and embedded systems.
Extends	None.
Attributes	None.
Associations	***AdaptiveMiddleware (exactly one)*** The adaptive middleware associated with the application domain.

Class Name	**QoS-Enabled**.
Semantics	QoS-oriented middleware supports real-time and multimedia applications, such as avionics systems, video conferencing and Internet telephony, that are required to meet deadlines and adhere to some QoS contracts, which define the acceptable levels of QoS.
Extends	***ApplicationDomain***.
Attributes	None.
Associations	None.

Continued on next page

Table A.1 – continued from previous page

Class Name	**Dependable**.
Semantics	Dependable middleware supports critical distributed applications that are required to be correctly operational, such as military command and control and medical applications.
Extends	***ApplicationDomain***.
Attributes	None.
Associations	None.
Class Name	**Embedded**.
Semantics	Embedded middleware supports applications that are required to have small footprints to be able to run on very limited memory devices, including set-top boxes, smart phones, hand-held devices, industrial controllers, and scientific instruments.
Extends	***ApplicationDomain***.
Attributes	None.
Associations	None.
Class Name	**ContentTransferProtocol (abstract)**.
Semantics	A protocol defining the rules and implementation details of attaching and transferring content along with messages.
Extends	***Protocol***.
Attributes	None.
Associations	None.
Class Name	**PackageContentSelectionPolicy**.
Semantics	The selection policy defining the service selection rules.
Extends	None.
Attributes	None.
Associations	***PackageContent (zero or more)*** The element(s) involved in the policy.
Class Name	**PostConditions**.
Semantics	The post-conditions of a service. The post-conditions describe the effects of the service after it is executed. Although the post-conditions are part of the semantic description of a service, since they are not included in our framework they were added in this diagram.
Extends	None.

Continued on next page

Table A.1 – continued from previous page

Attributes	None.
Associations	None.

Bibliography

[1] AndroMDA 3.2. Available at http://www.andromda.org.

[2] openArchitectureWare (oAW). Available at http://www.openarchitectureware.org/.

[3] D. H. Akehurst, W. G. Howells, and K. D. McDonald-Maier. Kent Model Transformation Language. In *Proceedings of Model Transformations in Practice Workshop, MoDELS Conference, Montego Bay, Jamaica*, 2005.

[4] Rama Akkiraju, Joel Farrell, John Miller, Meenakshi Nagarajan, Marc-Thomas Schmidt, Amit Sheth, and Kunal Verma. Web Service Semantics - WSDL-S. November 2005. Available at http://www.w3.org/Submission/WSDL-S/.

[5] Eric Armstrong, Jennifer Ball, Stephanie Bodoff, Debbie Bode Carson, Ian Evans, Dale Green, Kim Haase, and Eric Jendrock. The J2EE 1.4 Tutorial. December 2005. Available at http://java.sun.com/j2ee/1.4/docs/tutorial/doc/.

[6] Bill Burke, Sacha Labourey, and Richard Monson-Haefel. Enterprise JavaBeans, 4th Edition. OReilly, June 2004.

[7] Ethan Cerami. Web Services Essentials, Distributed Applications with XML-RPC, SOAP, UDDI & WSDL. OReilly, February 2002.

[8] S. Chaiyakul, K. Limapichat, A. Dixit, and E. Nantajeewarawat. A Framework for Semantic Web Service Discovery and Planning. In *Cybernetics and Intelligent Systems, 2006 IEEE Conference*, pages 1–5, June 2006.

[9] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. March 2001. W3C Specification.

[10] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. In *IBM Systems Journal*, volume 45, June 2006.

[11] A. D'Ambrogio. A Model-driven WSDL Extension for Describing the QoS of Web Services. In *Web Services, 2006. ICWS '06. International Conference*, pages 789–796, September 2006.

[12] Jos de Bruijn, Christoph Bussler, John Domingue, Dieter Fensel, Martin Hepp, Uwe Keller, Michael Kifer, Birgitta Knig-Ries, Jacek Kopecky, Rubn Lara, Holger Lausen, Eyal Oren, Axel Polleres, Dumitru Roman, James Scicluna, and Michael Stollberg. Web Service Modeling Ontology (WSMO). June 2005. Available at `http://www.w3.org/Submission/WSMO/`.

[13] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. In *Human-Computer Interaction (HCI) Journal*, volume 16, 2001.

[14] El-Sayed and Black. Semantic-based context-aware service discovery in pervasive-computing environments, IEEE International Workshop on Services Integration in Pervasive Environments. 2006.

[15] Wang et al. A QoS selection model for semantic web services, ICSOC. 2006.

[16] Joel Farrell and Holger Lausen. Semantic Annotations for WSDL and XML Schema. April 2007. Available at `http://www.w3.org/TR/sawsdl/`.

[17] D. Fensel and C. Bussler. The Web Service Modeling Framework WSMF. In *Electronic Commerce: Research and Applications*, pages 113–137, 2002.

[18] M. Fowler. *UML Distilled*. Addison-Wesley, 2005.

[19] Andreas Friesen and Kioumars Namiri. Towards Semantic Service Selection for B2B Integration. In *Workshop proceedings of the sixth international conference on Web engineering ICWE '06*. ACM Press, July 2006.

[20] T. Gagnes, T. Plagemann, and E. Munthe-Kaas. A Conceptual Service Discovery Architecture for Semantic Web Services in Dynamic Environments. In *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference*, pages 74–83, 2006.

[21] Object Management Group. Common Object Request Broker Architecture: Core Specification. March 2004. Available at `http://www.omg.org/technology/documents/formal/corba_iiop.htm`.

[22] Object Management Group. Enterprise Collaboration Architecture (ECA) Specification. Frebruary 2004. Available at `http://www.omg.org/cgi-bin/doc?formal/2004-02-01`.

[23] Object Management Group. MOF QVT Final Adopted Specification. November 2005.

[24] Object Management Group. UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. May 2006. OMG Available Specification.

[25] Thomas Gruber. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *International Journal Human-Computer Studies*, 43:907–928, November 1995.

[26] V. Haarslev and R. Moller. Description of the Racer System and its Applications. In *Int. Workshop on Description Logics (DL-2001), Stanford, USA*, August 2001.

[27] Martin Hepp. Semantic Web and semantic Web services: father and son or indivisible twins? *Internet Computing, IEEE*, 10:85–88, March-April 2006.

[28] Juanjuan Jiang and Tarja Systa. UML-Based Modeling and Validity Checking of Web Service Descriptions. In *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference*, July 2005.

[29] Simon Johnston. UML 2.0 Profile for Software Services. April 2005. Available at `http://www-128.ibm.com/developerworks/rational/library/05/419_soa/`.

[30] F. Jouault and I. Kurtev. Transforming Models with ATL. In *Proceedings of Model Transformations in Practice Workshop (MTIP), MoDELS Conference, Montego Bay, Jamaica*, 2005.

[31] Maksym Korotkiy and Jan Top. Onto-SOA: From Ontology-enabled SOA to Service-enabled Ontologies. In *Telecommunications, 2006. AICT-ICIW '06. International Conference on Internet and Web Applications and Services/Advanced International Conference*, pages 124–130, February 2006.

[32] Julia Kuck and Melanie Gnasa. Context-Sensitive Service Discovery Meets Information Retrieval. In *Pervasive Computing and Communications Workshops, 2007. Per-*

Com Workshops '07. Fifth Annual IEEE International Conference, pages 601–605, March 2007.

[33] KangChan Lee, JongHong Jeon, WonSeok Lee, Seong-Ho Jeong, and Sang-Won Park. QoS for Web Services: Requirements and Possible Approaches. November 2003. W3C Working Group Note.

[34] Anne Thomas Manes. Enabling Open, Interoperable, and Smart Web Services, The Need for Shared Context. March 2001. Available at `http://www.w3.org/2001/03/WSWS-popa/paper29`.

[35] Anbazhagan Mani and Arun Nagarajan. Understanding quality of service for Web services. January 2002. Available at `http://www-128.ibm.com/developerworks/library/ws-quality.html`.

[36] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. OWL-S 1.1 Release Technical Overview. November 2004. Available at `http://www.daml.org/services/owl-s/1.1/overview/`.

[37] Maximilien and Singh. A framework and ontology for dynamic web services selection, IEEE Internet Computing. 2004.

[38] Dave Menendez. Where Am I Language (WAIL). 2002. Available at `http://www.eyrie.org/ zednenem/2002/wail/`.

[39] Microsoft. Overview of the .NET Framework. Available at `http://msdn2.microsoft.com/en-us/library/a4t23ktk(VS.80).aspx`.

[40] Sun Microsystems. Java Web Services Developers Pack (WSDP). January 2002. Available at http://www.xml.com/pub/r/1315.

[41] P.A. Muller, F. Fleurey, and J.M. Jézéquel. Weaving Executability into Object-Oriented Metalanguages. In *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica*, pages 264–278, 2005.

[42] Hyun Namgoong, Moonyoung Chung, Kyung il Kim, HyeonSung Cho, and Yunku Chung. Effective semantic Web services discovery using usability. In *Advanced Communication Technology, 2006. ICACT 2006. The 8th International Conference*, volume 3, February 2006.

[43] University of Paderborn Software Engineering. Fujaba Tool Suite 5. Available at http://wwwcs.uni-paderborn.de/cs/fujaba/.

[44] R. Pompa. Java Emitter Templates (JET) Tutorial. May 2004. Available at http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html.

[45] Jinghai Rao, D. Dimitrov, P. Hofmann, and N. Sadeh. A Mixed Initiative Approach to Semantic Web Service Discovery and Composition: SAP's Guided Procedures Framework. In *Web Services, 2006. ICWS '06. International Conference*, pages 401–410, September 2006.

[46] S. Russell and P. Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall, 1995.

[47] S. M. Sadjadi and P. K. McKinley. A survey of adaptive middleware. December 2003. Technical Report MSU-CSE-03-35, Computer Science and Engineering, Michigan State University.

[48] Bran Selic. Model-Driven Development: Its Essence and Opportunities. In *Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium*, 2006.

[49] Evren Sirin and Bijan Parsia. The OWL-S Java API. 2004. Available at `http://iswc2004.semanticweb.org/posters/PID-CUIIDZKF-1090286595.pdf`.

[50] N. Sriharee. Semantic Web Services Discovery Using Ontology-Based Rating Model. In *Web Intelligence, 2006. WI 2006. IEEE/WIC/ACM International Conference*, pages 608–616, December 2006.

[51] N. Srinivasan, M. Paolucci, and K. Sycara. CODE: A Development Environment for OWL-S Web services. In *3rd International Semantic Web Conference*, 2004.

[52] N. Srinivasan, M. Paolucci, and K. Sycara. Semantic Web Service Discovery in the OWL-S IDE. In *System Sciences, 2006. HICSS '06. Proceedings of the 39th Annual Hawaii International Conference*, volume 6, January 2006.

[53] K. Sycara, S. Widoff, M. Klusch, and J. Lu. LARKS: Dynamic Matchmaking among Heterogeneous Software Agents in Cyberspace. In *Autonomous Agents and Multi-Agent Systems*, pages 173–203, 2002.

[54] Katia Sycara, Massimo Paolucci, Anupriya Ankolekar, and Maveen Srinivasan. Automated discovery, interaction and composition of Semantic Web services. In *Elsevier Journal of Web Semantics*, pages 27–46, 2003.

[55] J.T.E. Timm and G.C. Gannod. A model-driven approach for specifying semantic Web services. In *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference*, volume 1, pages 313–320, July 2005.

[56] A Min Tjoa, Amin Andjomshoaa, Ferial Shayeganfar, and Roland Wagner. Semantic Web challenges and new requirements. In *Database and Expert Systems Applications, 2005. Proceedings. Sixteenth International Workshop*, pages 1160–1163, August 2005.

[57] Kim Topley. Java Web Services in a Nutshell. OReilly, June 2003.

[58] D. Varró and A. Pataricza. Generic and Meta-Transformations for Model Transformation Engineering. In *Proceedings of the 7th International Conference on the Unified Modeling Language, Lisbon, Portugal*, pages 290–304, 2004.

[59] Larry Yusuf, Mandy Chessell, and Dr. Tracy Gardner. Implement model-driven development to increase the business value of your IT system. January 2006. Available at `http://www-128.ibm.com/developerworks/library/ar-mdd1/`.